Security through Network-wide Diversity Assignment

A Thesis

Submitted to the Faculty

of

Drexel University

by

Adam J. O'Donnell

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

September 2005

Drexel University
Office of Research and Graduate Studies
Thesis Approval Form
(For Masters and Doctoral Students)

Hagerty Library will bond a copy of this form with each copy of your thesis/dissertation.

This thesis, entitled Security through Network-wide Diversity Assignment

and authored by Adam J. O'Donnell, is hereby accepted and approved.

Signatures:

Chairman, Examining Committee: [illegible]

Supervising Professor: [illegible]

Committee Members:

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Program Advisor: [illegible]

Department Head: [illegible] 9/23/05

Dedications

This dissertation is dedicated to my parents, Joseph and Monica, for instilling in me the thirst for knowledge that has led me to this point, and to my wife Sophy for providing me with the love and support I needed to see me through.

Acknowledgments

I have fantasized about writing this page of my dissertation for many years. Now that the time has come I find myself struggling to compose the words which reflect how those closest to me have guided my work over the years.

The path of research is long, twisting, and poorly lit. The trip would be far more difficult if it weren't for the many who have previously trodden the path and the few who directly support you on your journey. I owe all of those who have walked along my side for these years my deepest gratitude.

Drexel University has been my home for the past nine years. From the moment I arrived here, members of the faculty and staff of the College of Engineering have provided me with much needed counsel. While some are no longer with the university and a few are no longer of this world, they all deserve to be mentioned here, particularly Bob Quinn, Allen Rothwarf, Vaughn Adams, Afshin Daryoush, Maja Bystrom, Athina Petropulu, Harish Sethu, Vassilis Prevelakis, Steven Weber, Bob Koerner, and Mun Choi. I additionally want to thank the entire faculty and staff of the Electrical and Computer Engineering Department, especially Nihat Bilgutay, Kathy Bryant, Stacey Caiazzo, Dolores Watson, and Tanita Chapelle, for building the learning environment in which we all share.

It is difficult to separate your image of self from the research you produce while a graduate student, but the commiseration provided by colleagues helps you remember that you are a person and not just your work. I would like to thank all my current and previous labmates, particularly Madhusudan Hosaagrahara, Valli Rammohan, and Michelle Sipics for lending an ear and and eye to my work. I would also like to thank the members of what became my defacto surrogate lab in the Computer Science department for allowing me to routinely pick their brains, namely Ali Shokoufandeh, Jeff Abrahamson, M. Fatih Demirci, and Trip Denton.

Many a day's discussions of work turned into a night's discourse on philosophy and life in general, which in turn made graduate school a far more enjoyable experience. I don't know how I would have made it through without the emotional support provided by these cathartic evenings. For this, Nick Kirsch and Eric Gallo, I owe you an incredible debt.

My education has never been limited to the classroom, and my contacts in the security industry have heavily influenced my work and development as a professional. In particular, I would like to thank Ralph Logan for showing me the ropes of the business world, and Tim Newsham and Jose Nazario for setting an example of how an engineer and an academic can thrive in industry.

Completing my graduate work would not have been possible without financial support from three sources. I would like the ECE Department for providing the Colehower Fellowship during my critical first year, and the fine people at the National Science Foundation for the Graduate Fellowship Research Program[†] that sponsored the bulk of this dissertation. Finally, I again extend my gratitude Bob Koerner and the Koerner Family for providing both funding through the Koerner Family Fellowship as well as career guidance.

The final gatekeepers to the PhD are my dissertation committee, who graciously donated their time for this pursuit. I would like to thank all of them for their time invested in reading my work and attending my presentation.

There were a few points during my work that I thought the final goal was unattainable. My advisor, Harish Sethu, has helped shape my work and development as a researcher from my senior year as an undergraduate to this point. More importantly, he never lost faith in me even when I risked losing faith in myself. Without mincing words, I could not have asked for a better advisor. Thank you for everything.

Finally, without the support and love of my family, none of my education would be

[†]This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

possible. Thank you Mom and Dad for the trips to the libraries and museums, the books and magazines, and the nurturing and stimulation you provided that made it possible to me to reach this goal.

And last but not least, without you Sophy, completing my education would have been impossible. I love you, and thank you for everything you have done for me.

Table of Contents

List of Tables

List of Figures

Abstract

Security through Network-wide Diversity Assignment
Adam J. O'Donnell
Harish Sethu, Ph.D.

The best efforts of the computer security community have not eliminated software with hidden attackable vulnerabilities in the world. Code analyzers and hardened operating environments have reduced software bugs. Improved training has created capable security administrators who have decreased the population of exploitable systems through attentive patching and network access control. A third approach to combating vulnerabilities has been proposed which requires the use of diverse software packages to slow or stop attackers. Literature examining the topic of *software diversity* details a variety of implementations, but for both business and technical reasons, the limited number of functionally equivalent yet distinct software packages makes diversity a less effective strategy than one may like.

In this dissertation, we make diversity a viable security strategy despite the limited number of diverse systems. We abstract the software diversity concept to a hypergraph by considering how techniques for generating diversity interact and present themselves to attackers. We show that diversity's utility can be increased through the use of graph coloring algorithms. We design a series of distributed graph coloring algorithms and test these on real-world graphs collected from the BGP topology of the IPv6 backbone and nine months of e-mail traffic. The diversity assignments are quantified through the use of graph theory-based metrics, such as the monochromatic edge count and the disconnected component count, as well as the epidemic threshold, a metric borrowed from epidemiology research.

Any methodology for increasing the attack tolerance of a network is destined to come under attack itself. We examine the tradeoff between the quality of our algorithm's diversity assignment produced and our algorithm's attack tolerance. We show that the attack tolerance of our algorithms can be increased by presenting an attacker with a diversity of graph coloring algorithms. Based upon our observations, simulations, and analysis we are left with a confirmation of our thesis: not only is diversity critical for improving the attack tolerance of a network, but diversity must be applied at *all* levels of system design including mechanisms to introduce the diversity itself.

Chapter 1. A Brief Overview of Computer Security

Basic research is what I'm doing when I don't know what I'm doing.

Werner von Braun

1.1 Introduction

The connected nature of our modern computer systems has given us unparalleled access to information and communication resources. The connected nature of our systems has also given unparalleled access to computer crackers, who, both directly and through *viruses* and *worms*, compromise the integrity of our electronic assets. As of the time of the writing of this document, computer security issues have become a routine and unwelcome event in every user's life. While the majority of users are not adept at using computer access control features to their full capacity [92], attackers rarely attempt brute force attacks against advanced access control systems. Instead, they invest time in developing attacks against potential security issues created by coding and configuration problems [12], referred to collectively as *vulnerabilities*.

All software, regardless of authorship, may harbor hidden security related vulnerabilities, and these vulnerabilities will eventually be discovered. Upon the discovery of a vulnerable situation in a piece of software, code to attack the vulnerability, known as an *exploit*, may be created and remain secret for many months as it makes its rounds through the hacker community[1]. The instant the exploit is detected by the computer security community, it acquires the *Zero Day* monicker, which reflects the exploits novelty to the software industry. The appearance of a zero day exploit drives the maintainer of the vulnerable code to release a patch to the software to correct the issue. Responsibility for final remediating

[1]An excellent guide on writing software exploits can be found in [29].

of the issue returns to the global population of system administrators and users, who must install the patch to correct the issue. Studies have shown that both the *zero day* and the *unpatched* period can be unexpectedly long, with the unpatched period lasting years [4]. Furthermore, an individual's diligence in system maintenance is insufficient for protecting his or her access to data; the volume of unmaintained systems on the Internet places everyone's access to data at risk [79].

The current state of computer security dictates that users are either waiting to be attacked by a new exploit for previously unknown vulnerabilities or by an old exploit for an unpatched vulnerability. This dualism provides two natural avenues for research endeavors and security products. The first course of action deals with the reduction of security vulnerabilities due to known classes of programming and configuration bugs, while the second course focuses on the management of large groups of systems with known vulnerabilities. The fields of vulnerability reduction and mitigation are both vast, and it would be unwarranted to present a complete survey of both topics in this dissertation. Instead, a brief summary of the core research directions in both areas is presented below.

1.2 Combating Vulnerabilities Before Discovery

A vast majority of the exploits utilized by computer hackers arise out of programming flaws created by the application developers. For example, classic buffer overflow vulnerabilities [20] are created when a programmer writes data to memory addresses located in the stack without confirming that the data will fit inside the space previously allocated by the programmer. The overrun of data which flows out of the allocated space can rewrite the instruction pointer of the calling function; if the overwriting data is carefully chosen, the stored instruction pointer can be redirected back into the data block which was copied into the buffer. As soon as the current function exits, the next instruction fetched will be code which exists on the stack rather than inside the program's data segment. If the data being passed to the program is completely controlled by the same user who owns the targeted program in memory, then there is no security issue. However, if the program executes at a higher level of privilege than the user who generates the data accepted by the program, then the user generating the data can hijack the program and execute instructions as if the user was at the same privilege level as the program.

These attacks have become so commonly exploited that they are considered to be cliché in the security community. The removal of the well-known classes of security vulnerabilities, such as buffer overflows, will involve developer education to reduce the number of new vulnerabilities. Additionally, it will require legacy code to be retrofitted to either reduce the exploitability of vulnerabilities or reduce the number of exploit targets. A general overview of software vulnerabilities, as well as proper coding techniques, can be found in [83].

Reducing the exploitability of bugs in software is referred to as *code hardening*. While these techniques do not remove the vulnerability completely, they do make the creation of exploits for particular vulnerabilities significantly more difficult. Hardening methods have been identified which have been shown to be somewhat effective against stack-based buffer overflow attacks [19, 23] and pointer manipulation [17]. An extensive analysis of publicly available hardening schemes for buffer overflow prevention produced by Wilander and Kamkar has shown them to be less than perfect [88] with the best tool stopping only 50% of studied vulnerabilities. The figures provided do not include countermeasure subversion techniques, such as the pointer-overwriting method discussed in *Phrack Magazine* [11]. A survey of compiler-level buffer overflow techniques, as well as countermeasures against them, was published in [2].

Code and kernel modification can completely prevent certain classes of vulnerabilities. For example, format string attacks [16] and race condition exploitation [18] can be eliminated from consideration by hackers in their current forms. Other classes of vulnerabilities, such as the aforementioned buffer overflow issue, can be reduced, but not completely eliminated, through static analysis[2]. Nevertheless, static checking of both standard code [56] and type-enhanced code has proven to be an effective method of decreasing the number of vulnerabilities.

While static checking of code may not be able to catch all memory-related software vulnerabilities, run-time bounds checking of memory access has also been examined. In general, this class of techniques works by keeping track of both statically and dynamically allocated memory and performing bounds checks on all memory access operations to confirm they fall inside “safe” memory blocks. Lhee and Chapin [80] proposed the use of a compiler extension which uses typing hints for the construction of an allocated memory table, which is then used for bounds checking in memory copy related bounds checking. Ruwase and Lam [75] pointed out that many out of bounds memory accesses are not security critical, and produced a bounds checking solution which allows for these cases. In general, all runtime bounds checking techniques available at the current time incur dramatic performance penalties.

Rather than attempting to squash all vulnerabilities in a single piece of software, researchers have tried to apply techniques borrowed from the fault tolerance community to allow software to tolerate attacks. Rinard *et al.* coined the term “failure oblivious” computing [73] to describe software which follows standard execution paths in the face of invalid memory access. Failure oblivious software is implemented using code which performs continuous checking and validation of memory and control flow, which are both schemes developed in the fault tolerance field.

[2]A trivial proof of the impossibility of eliminating all buffer overflows through code analysis was described by Larochelle and Evans, where they state that the issue is equivalent to solving the halting problem.

1.2.1 Fault Tolerance and N-Version Programming

As stated, the security community has been applying decades-old fault tolerance techniques developed for detecting defective systems and code to today's security problems. Avižienis first proposed the use of N-version programming [6], which suggests developing multiple functionally equivalent software packages from the same specification and then use a voting algorithm to choose the correct program output. While similar techniques have been used in hardware fault tolerance before, Avižienis' work is the first place where the technique was applied to software. Extensions of this work utilize a control flow monitor to compare the multiple simultaneously executing processes to detect deviations from standard behavior, rather than using a majority voting system that examines the final output.

The critical assumption made by N-version programming is that software packages developed from the same specification in a clean-room environment will possess uncorrelated errors. This assumption has been experimentally explored in [52]. The researchers found that it is possible to generate a set of diverse software packages from a single specification, but that the existence of faults in multiple versions of the same software package was not completely independent as individual programmers often made similar mistakes.

1.3 Combating Vulnerabilities After Discovery

It is not unreasonable to assume that software vulnerabilities, either created by programming or configuration flaws, are going to be a permanent fixture in the computing environment. Mitigation strategies for dealing with vulnerable systems which are in production must be considered as well. Ultimately, a software vulnerability can be removed only through the application of a *patch* [22], or code change, which removes the code or configuration flaw. The distributed and decentralized nature of desktop system maintenance places the ultimate responsibility of patch maintenance in the hands of the desktop's owners, and facilities have to be provided for both the detection and mitigation of publicly active vulnerabilities at the network level.

Detection of software vulnerabilities at the network level can be performed through either active or passive measurement of the network. The active detection of vulnerabilities, known in industry as both *vulnerability assessment* and *penetration testing* [58], requires a database of known vulnerable software packages along with a program which can test for the markers associated with the software in the database on each machine on the network. Passive vulnerability detection works by examining network traffic and performing string matching against known attack patterns. This role is performed by signature-based Intrusion Detection Systems, or IDS, such as Snort [74]. Vulnerability assessments detect issues which may have existed, and have been exploited, for some time; additionally vulnerabilities detected using signature-based IDSes are only found at the instant of exploitation. Therefore, both of these technologies are of limited use for vulnerability mitigation.

One of the earliest tools available for network-wide vulnerability mitigation has been network-edge packet filters, or firewalls [14]. Firewalls, famously described by Cheswick termed as espousing a "crunchy shell around a soft, chewy center" model of network security, are only a temporary countermeasure to newly discovered vulnerabilities. These devices do not prevent software vulnerabilities from being exploited, but they can be used to drastically limit the number of locations on the Internet from which an attacker can launch an exploit against the protected hosts, which in turn gives the system administrators time to apply patches when a vulnerability arises.

The vulnerability detection and mitigation principles discussed can be improved through a variety of means. For example, intrusion detection systems which look for anomalies in the behavior of network traffic rather than known-bad traffic signatures are being actively developed and deployed [25]. Firewall systems can be distributed throughout a network, rather than being placed at a single choke point, so that the number of nodes which can successfully launch an attack against any given protected system is reduced [9]. Additionally, a blend of intrusion detection and automated mitigation techniques, such as network intrusion prevention systems similar to Snort InLine, can be employed [44]. All post-vulnerability disclosure solutions, however, require an effective infrastructure to alert system administrators to new problems [57], perceptive security teams who are able to rapidly act on new security information, and mostly lazy system attackers.

The window on vulnerability exploitation is slowly being closed from both sides. Improved software quality assurance procedures, developer education, and automated tools are helping to reduce the number of easily exploitable vulnerabilities present in new software. Similarly, network administrators are acting on vulnerability announcements more quickly than in the past. It is unlikely that the reduction of vulnerabilities coupled with rapid mitigation techniques will ever eliminate remote exploitation; the prerogative of vulnerability disclosure still falls in the hands of the individual or team who discovers the issue. Given the impossibility of elimination of all software vulnerabilities before code release, the security community should expect to see the appearance of zero-day exploitation in the visible future. The persistent gap between reduction techniques and mitigation techniques discussed opens up another avenue for combating attacks.

1.4 The Software Diversity Compromise

We can draw two conclusions from the surveys presented in Sections 1.2 and 1.3. The analysis of vulnerability reduction techniques shows that all software, even after extensive examination, may harbor hidden vulnerabilities which have yet to be discovered. Additionally, vulnerability detection and mitigation techniques are only truly effective against vulnerabilities which have been publicly disclosed and patched. Any scheme which does not attempt to directly reduce vulnerabilities or stop vulnerabilities in the wild may be useful for combating attacks in this gap.

As pointed out in [47], remote attacks against software are produced to extremely tight specifications. Unlike most conventionally produced software, whose specification is laid out on paper, an installation of a vulnerable piece of software *is itself* the specification for an exploit. A network defender is able to implement a large number of software packages which meet the functional specifications required by the network, but deviate from the virtual specification created by the vulnerable software package previously targeted by the attacker. This is the essence of software diversity. By employing a *diversity of software and hardware packages* to serve the various needs of a network, an administrator is able to reduce the effectiveness of a single system-specific attack against systems under their control.

The use of diverse software systems to combat deliberate faults on a single system has existed for some time. For example, Joseph and Avižienis extended the N-Version Programming concept to defend against computer viruses on a single system [46]. The modern view of software diversity is not concerned with generating a large collection of diverse software packages for simultaneous execution on a single system, but with assuring that networks, taken as a whole, are comprised of a diversity of systems.

Evidence corroborating the inherent value of heterogeneity in a population can be found across a variety of fields, including the field of biology and organic systems [32]. The American farmer, for example, learned of the disastrous consequences of sowing a limited number of genetic strains and its subsequent vulnerability to an infectious agent of limited capability. In the 1970's, the U.S. corn crop was destroyed when the *Bipolaris Maydis* pathogen ate through the genetically similar plantings. This single event destroyed over $1 billion of harvestable corn, or about 15% of the crop [42].

Researchers have attempted to leverage the diversity through biology concept by building systems that directly emulate biological system behavior. The most illustrative example of this concept was described by George and his coworkers [37], where diverse sets of cellular automata work in concert to perform a computation.

The security community views diversity as being absent in today's networks, and has made this sentiment clear in public forums [35, 36, 39, 78]. While the software monoculture present may have been created by either market forces or by technical constraints, researchers have been studying methodologies and techniques which can be used to artificially generate diverse software packages.

1.5 Organization

The rest of the dissertation is organized as follows. In Chapter 2 we explore the methods of implementing software diversity, as multiple techniques for the artificial generation of diverse software packages exist. In Section 2.2, we sketch the basic premise of this dissertation: that the utility of using diverse software packages for slowing attackers can be increased by a careful assignment of diversity techniques to hosts and nodes on a network. We provide a unifying framework that allows for the abstraction and reasoning about these software diversity schemes in Chapter 3. Using this framework, we formally define a new class of problems that deal with choosing the right subset of diverse software packages and diversity generation techniques, known as the software diversity assignment problem. Upon examination of our model presented in Chapter 4, we have discovered that the amount of diversity required to slow an attacker can be far less than the number of hosts on a network, and instead the amount of diversity required is a function of the structure of the network that is being diversified. We then exploit this discovery for the design of distributed algorithms for assigning diversity on a network of systems. In Chapter 5 we both confirm that our diversity assignment strategies are effective in combating self-propagating malware using techniques borrowed from the field of computer virus modeling and simulation, and we show that the way to guarantee the security of our diversity assignment scheme is to introduce a diverse set of diversity assignment algorithms.

Chapter 2. The Software Diversity Compromise

It is time for parents to teach young people early on that in diversity there is beauty and there is strength.

Maya Angelou

2.1 Generating Diversity

In order to counteract the lack of diversity in the Internet, researchers have focused on the method of diversifying pre-existing architectures, source code, and binaries in order to artificially generate a diversity of software packages. In general, we can classify the points at which diversity can be applied into the following categories: *Requirements*, *Architecture*, *Implementation*, and *Realization*. While other classification schemes of diversity techniques have been presented [26], we are less interested in the managerial aspect of applying diversity to entire business processes, and more concerned with diversity implementation schemes.

During the *Requirements* phase, early design considerations which provide diverse methods of interacting with networked devices, processing information, and interacting with the user can be factored into the initial requirements document. Schemes which generate a loose functional equivalence between different binaries would be applied during this stage [94]. In a similar vein, the *Architecture* of the software architecture can be varied to allow for different data flows and process interaction, while still maintaining a standardized software interface.

The majority of the diversity schemes present in the literature consider how diversification can be applied during the *Implementation* and *Realization* phases of the software development cycle. The *Implementation* phase allows for source code to be modified in an algorithmic fashion, for the software to be built using different programming languages, and for the software to be built by independent teams of developers using the same language. As proposed by Forrest, Somayaji, and Ackley, automated techniques which manipulate source code by reordering source code, adding and removing non-functional code, or changing the linking order of dynamic libraries can be utilized [34].

Researchers working on preventing reverse engineering of binaries have developed code obfuscation techniques which can also be used to diversify software packages. A technique for obfuscating Java source code, which uses similar code reordering techniques proposed by Forrest, is presented in [15]. A general model of code obfuscation was developed by Wroblewski [90].

After code implementation, the final *Realization,* or build and execution, of the software can be modified through a wide variety of techniques, including the compiler-driven randomization techniques [31, 85]. In fact, many of the code reordering techniques which provide memory randomization functionality can be applied at runtime after a binary has been created [10, 91].

At the final stage of development, the instruction set used can be diversified without a wholesale switch of system architectures. Barrantes *et al.* proposed the use of an x86 to x86 translator to randomize a system binary before being run on a virtualized x86 processor with an obfuscated instruction set [8]. In related work, Kc, Keromytis, and Prevelakis suggested the use of XOR encryption of the instruction set at the processor level to produce a the same instruction obfuscation effect [48]. Both systems serve the same purpose by converting maliciously injected code into binary strings which have little meaning for the processor. Additionally, both techniques are not without practical precedent, as a similar technique was proposed by Cowan *et al.* for protecting pointers in memory [17]. Both forms of artificial instruction set randomization appear to be broken [77], however, due to irregularities in the byte size of each opcode present in the x86 platform.

The code reordering and reforming techniques are expanded upon in [15] for the purpose of obfuscating Java code against reverse engineering. Wang and her coauthors [85] describe code modification techniques for use in protecting high-availability mechanisms which are currently employed in server systems.

The compile-time techniques discussed are readily available for download, and have found their way into open source operating system distributions [24]. Address space randomization is implemented in the Linux PaX toolkit [68], and compile time randomization of stack offsets has been implemented in GCC [31]. It has been pointed out that address space randomization doesn't work as well as predicted in architectures with smaller address spaces due to the fact that large portions of the address space are reserved by the operating system, and are not accessible for user-land memory addressing [76].

2.2 The Case for Assigning Diversity

The attacks discussed against the publicly available diversity generation techniques [76, 77] undermines the assumption that a diverse pool of software can be created at a low cost. Furthermore, an analysis of POSIX-compliant operating systems showed that faults were highly correlated across different vendor's platforms, with the majority of common faults existing in upper-level functionality, such as C libraries. In general, as we descend from the high level components of a system through the core and into the original architecture specifications, software diversity becomes both more expensive to implement, and more effective against common faults. *We are forced to conclude that the cost of generating a set of truly diverse software packages makes diversity a scarce resource which must be carefully and consciously allocated in order for it to be maximally effective against attackers.*

For a single host, choosing the optimal set of diversity techniques and diverse software packages resolves down to a problem of economics. The benefit side of the equation consists of creating a system which is different enough from the global population of computers that an attack against any one system would be difficult to port to be effective against the diversified system. Each of the diverse software packages, source level, and compiler-driven diversity techniques have a associated cost figure, as they either cost money to purchase, decrease computing speed, or increase the amount of administration time required for patching and general system maintenance.

The burden of creating a host which is considered to be diversified as compared to all other hosts on the Internet is massive, but it is not one faced by a network administrator who has control over a large pool of systems. The network administrator's diversification task is not equivalent to solving the single host diversification problem for every machine on their network. Unlike the single host's administrator, a network administrator is able to leverage the restrictions placed on an attacker by the network topology in order to reduce the number of diverse software packages necessary. This is the fundamental thesis of our work: *by taking the topology presented to an attacker into account, an assignment of a small number of diverse software systems can be formulated which can slow or stop an attacker in their tracks* [63].

While it may be argued that the network topology traversed by an attacker is a complete graph, and every machine must be made diverse and separate from every other machine on the network, this statement is not true even for IP-level connectivity. The prevalence of firewalls and private address spaces prevent any machine from connecting to any other machine on the Internet. Furthermore, not every attack exploits IP-level connectivity for propagation. Worms which spread by traversing individual e-mail address books move through a network topology which is remarkably sparse [28], and client-server file sharing worms inhabit graphs which are largely bipartite [40].

2.2.1 Examples of Network Diversity Assignments

E-Mail Topologies: Any individual that utilizes e-mail has become a target of self-propagating code. Vulnerabilities associated with the default configurations of MIME han-

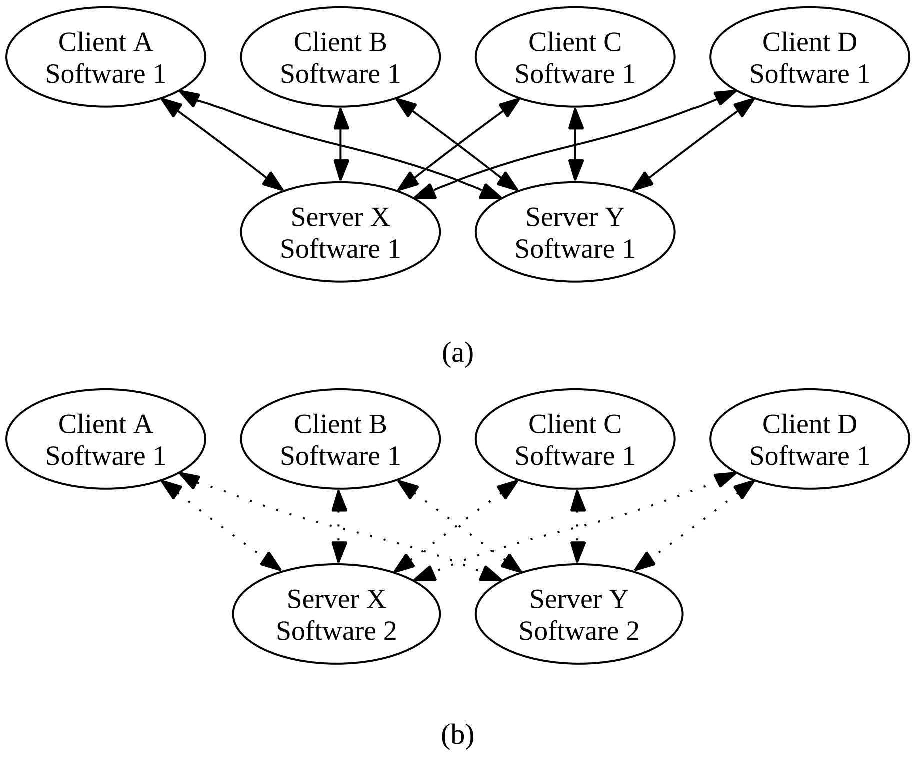
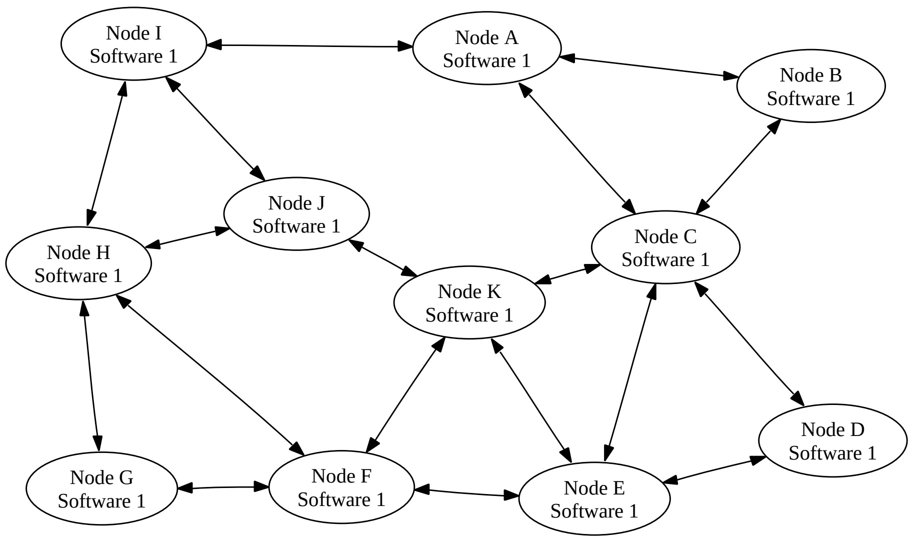


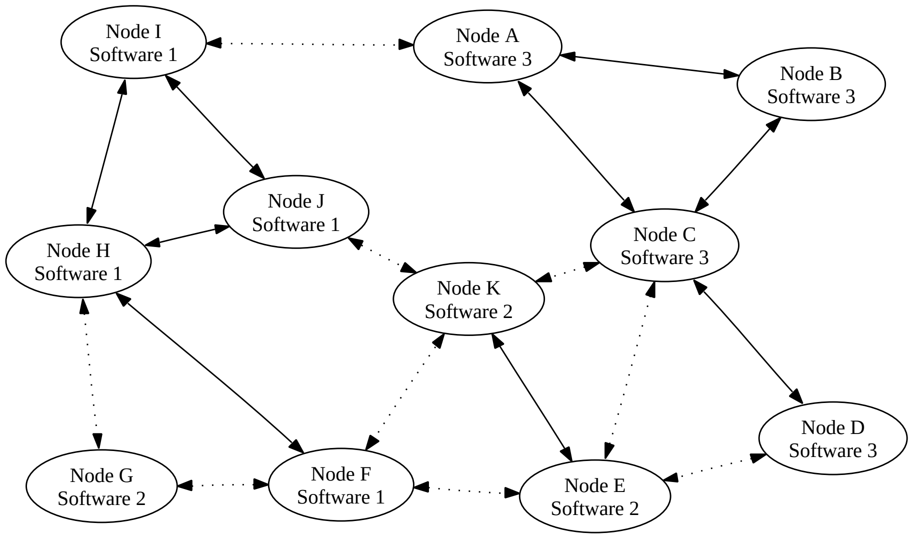
Figure 2.1: Comparison of network topologies utilizing either a single software package or a diverse software distribution. The effect of optimally distributing two software packages on a bipartite network is clear in (a) and (b). Bipartite networks such as these are often found in client-server file sharing topologies.

dlers [41] have given rise to client-side computer viruses [40]. Errors in the parsing code in major mail transfer agents have resulted in server-side attacks that are also propagated via e-mail traffic [55]. Secure diversity can be implemented in the stated situation through the utilization of interchangeable MIME and e-mail header parsers which are selected by the application based upon a topology-sensitive algorithm. Replacing one parser library with another would have no user-discernible impact on the software's behavior and performance.

Client-Server File Shares: Network-accessible file shares have become a popular target for platform-dependent worm propagation [43]. In many office environments, the file shares are partitioned into the client and server groups as shown in Figure 2.1(a), where



(a)



(b)

Figure 2.2: An additional comparison of network topologies utilizing either a single software package or a diverse software distribution. A random network topology clearly benefits from a random distribution of three heterogeneous software packages (a) as compared to a uniform distribution of a single package (b). While the assignment is sub-optimal, the number of edges which exist between nodes running similar software packages is clearly reduced.

communication links between similar systems are represented by a solid line. This partitioning can be enforced using firewalls and ACLs. A worm infection on a client system would be able to self-propagate to any machine in the file-sharing topology by first attacking a server machine; likewise, a worm infection on a server would have to first attack a client before propagating further.

The secure diversity principle can be quite effectively applied to such a network with only two different software packages. All previous communication links between similar systems are replaced by links between dissimilar computers, represented by the dotted lines in Figure 2.1(b). By utilizing a second software package for file sharing on the server systems, it is possible to prevent a client system from propagating a worm that attacks a vulnerability in the file sharing subsystem.

Sensor Networks: The networking field that would benefit greatly from the secure diversity principle is sensor networks [30]. Enforcing a diversity policy in a sensor network is less of an administrative challenge, since these large networks of relatively simple computational and environmental monitoring nodes are usually controlled by a single entity, be it a military commander or a building supervisor. Because the hardware is characterized as being relatively simple, it is not a major technical challenge to recreate their comparatively small software suite for the purposes of introducing variation between individuals in the population.

Consider the possibility of a system-wide vulnerability that allows for an attacker to take over a single networked sensor. A single attack can be used to leap-frog from node to node across the entire network, as indicated by the bidirectional links in Figure 2.2(a). Sensor networks can be distributed with multiple operating systems in ROM. After being dropped into the operational location, a node can load up one of a multiple set of OSes. By constructing a network that contains a multiplicity of operating systems, a single operating system-specific attack will not be able to propagate across the entire breadth of the network. Such a randomized distribution of software packages, as shown in 2.2(b), can reduce the number of possible node-to-node movements by an attacker.

2.2.2 Reasoning about Diversity

While the concept of diversity assignment schemes may be philosophically appealing, currently there is no formal system available for reasoning about diversity assignments. In the following chapter, we provide a framework that abstracts both the generation and attacking of diverse software packages.

Chapter 3. Generalizing and Assigning Software Diversity

Monocultures, like a field of corn, are susceptible to infections, but genetically diverse cultures, like a prairie, are extremely robust.

Neal Stephenson

3.1 Introduction

In this chapter, we create a generalized framework for classifying and analyzing diversified software that is driven not by the diversity schemes themselves but by how diversified software appears to an attacker. Consider the pedagogical example shown in Figure 3.1. A single system can be diversified by running an operating system variant, such as Linux or OpenBSD. Any single implementation inside the diversified set can then be diversified again by running the system on a different base hardware platform, namely either x86 hardware or SPARC hardware. If an attacker has a working exploit against software running on a Linux x86 system that they wanted to use against an OpenBSD/SPARC system, the attacker would have to mutate the attack so that it is effective against both a different operating system and a different platform.

In general, each diversity technique applied to a single system creates a pool of diverse systems from the originating system. Each system inside that pool can then be diversified by a separate technique to create additional instances of diverse software. This concept forms the basis of our diversity model. We consider every possible instance of software that can be generated by the application of diversity techniques, then place the software instances into the same set if they appear to an attacker as if they are separated by a single diversity technique. A single piece of software can be in multiple sets, as it can be used as a seed for multiple different diversity techniques.

The software instances are the vertices of a *hypergraph*, with the sets of diversified

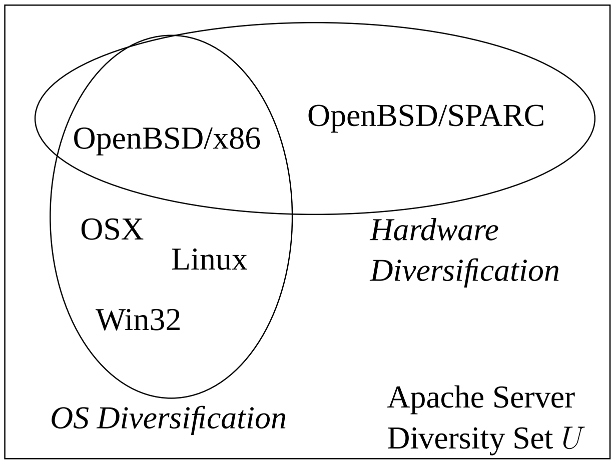


Figure 3.1: In the above, a single Apache installation is diversified by the introduction of different operating systems. One of the instances is further diversified by the introduction of multiple hardware platforms. An abstraction of the model is presented in Section 3.2.1.

variants generated by a single diversity technique forming the *hyperedges*. Since the hyperedges naturally overlap at points where a software package is diversifiable using more than one technique, we are able to reason about the use of multiple diversity techniques on a single software package. We can abstract the behavior of combining multiple diversity techniques as being a walk across intersecting hyperedges on the hypergraph. To an attacker, the amount of work that he or she must undertake in order to modify an effective exploit against one system so that it can compromise another is a function of the number of hyperedges, or diversity techniques, which separate the two instances of the software. In Section 3.2.2, we describe how metrics which derive from attack and defense modeling can be applied to the hyperedges for purposes of determining an optimal balance of attack tolerance and implementation cost. We examine what is effectively a trivial application of the model by examining the application of diversity techniques to a single system briefly in Section 3.3.1, and in more depth in Appendix A. The remainder of the dissertation is spent examining the problem of assigning diverse software packages to networks of systems, as described in Section 3.3.2.

3.1.1 Related Work

Our model is generated by examining how diversity appears to an attacker, and can easily be extended to encompass new diversity techniques and new forms of attacks. The generation of a diversity hypergraph is not dependent upon taxonomies of previously developed diversity techniques [50]. The generation of a diversity hypergraph for a real system may in fact lead to new forms of diversity taxonomies, ones where the effect of diversity on an attacker is central to the taxonomy. Taxonomies of attack techniques and methodologies [54] would potentially be useful for modeling the abilities of an adversary who is confronted with diversity techniques.

3.2 Defining the Diversification Hypergraph

Definition 1 *Let $d \in D$ be a single diversity technique in the set D of all diversity techniques. Let $u \in U$ be a single binary in the set U of all possible software binaries.*

The application of a single diversity technique in D takes a single instance of software in U and generates a set of software packages. The software generated by a single diversity technique is viewed to be interchangeable with one another as defined by the bounds of the diversity technique. Elements of the set of diversified software packages can be grouped together into equivalence classes, where any element in the class can be mutated to become another element in the class using a single diversity technique. For example, if the diversity technique requires systems to run separate operating systems, the set of diverse systems are equivalent under the bounds of operating system diversification.

Definition 2 *The elements $d \in D$ form* equivalence relations *on the set U. Two software packages $u_1, u_2 \in U$ are said to be* equivalent *under diversification scheme d if the only* *difference between the two elements in U results from the application of the diversity technique described by d.*

Definition 3 *Each equivalence relation generated by the elements of D creates* equivalence classes *over the elements contained in U. We denote the equivalence created by technique d between the two elements u_1 and u_2 in U as $u_1 \equiv_d u_2$.*

We can loosely classify the equivalence classes into several categories. Two software packages lie in a *binary equivalence class* if the lowest cost modification required to transform one software package into the next can be done at the binary level. If a vulnerable software package is diversified via a binary technique, the original attack target will still exist; the exact memory location of the attack target becomes far harder to find, however, due to the increased space over which the memory location of the attack target may exist. An example of a binary-level modification would be the randomization of the layout of a program and its linked libraries in memory [10, 34].

While it is possible to convert one program to another through bitwise adjustments, the process of doing so may be extremely time consuming. It could be far easier do make the modifications at the source code level and allow the compiler to generate the different binary. Likewise, if it becomes less costly to convert one binary package to another via source code modification than recompilation, then the two software packages lie in a *source equivalence* class created by the diversity technique. Attacks against software packages which have undergone a source modification technique must be modified themselves to be made effective against the newly diversified software packages. The modifications for the attack code may be as simple as a single modification in the attack binary, but given the stage of the development cycle at which the diversity technique is introduced, it is likely that a more advanced algorithm or manipulation scheme would have to be utilized for the attacker to successfully attack the diversified software package.

As stated in the introduction, several diversification strategies exist for the algorithmic modification of source code, such as adding or deleting nonfunctional code, code reordering, and randomizing memory layouts [34]. More invasive techniques which modify data and control flow are also feasible [85]. Incidentally, these code reordering and reforming techniques can also be effective against reverse engineering attacks [15].

Two software packages may have extremely differing lineage or development histories but serve the exact same purpose in a system. If two software packages provide effectively the same functionality, such as two distinct flavors of UNIX, then the software lies in a *functional equivalence class*. Probably the most studied method of generating functionally equivalent software packages uses the N-Version Programming technique [6, 46] discussed in Section 1.2.1.

The equivalence class generated by any given diversity technique may not be directly tied to the stage in the development cycle at which the diversity technique was applied. For example, consider a source code modification technique that works by repositioning variable declarations. The effect on the final diversified binaries that result from the technique's application can also be generated by directly modifying a compiled binary's memory structure [91]. If a system designer uses both the source and binary-level modifications, all the binaries generated using the diversity techniques would reside in the same equivalence class. The use of multiple diversity techniques on the same piece of software is described further in the following section.

3.2.1 Composition of Diversity Techniques

Definition 4 *A* composition *of diversity techniques is the serial application of the techniques one by one in order of temporal precedence.*

Composition increases the amount of work necessary to convert an attack which is effective against one software package to be effective against another one generated from the first via a set of diversity techniques. For example, a binary can be diversified using a compile-time memory space randomization scheme [91], then executed on a system which utilizes an encrypted instruction set [48]. Any attacker who wishes to take an attack which is effective against a single binary and mutate it so that it is effective against a binary which has undergone diversification using both techniques discussed would have to simultaneously de-randomize the memory space and decrypt the instruction set utilized by the diversified binary. An attacker may not need to manipulate an attack to solve two diversity techniques at the same time; if N-version programming is employed in the selection of the base operating system employed to run the binaries, an attacker can first solve all the mutations necessary to combat the introduction of the foreign operating system and then solve the issues associated with the instruction set and memory space manipulations.

Definition 5 *The set of all equivalence classes created by the diversity techniques in D form the hyperedges $\mathcal{E}$, which along with the elements of U define the* diversification hypergraph $H = (U, \mathcal{E})$.

In order to define properties about interactions between hyperedges, being able to identify individual hyperedges becomes a necessity. It is easy to see that every hyperedge in $\mathcal{E}$ can be identified by a diversity technique in D and a single binary in U which lies in the hyperedge. Consider two hyperedges which are created by the same diversity technique and containing the same binary. The diversity techniques from each hyperedge would create two equivalence classes that cover all binaries separated by the single diversity technique. Since both diversity techniques are identical, they would create equivalence classes which contained the same set of elements, and thus create the same hyperedge.

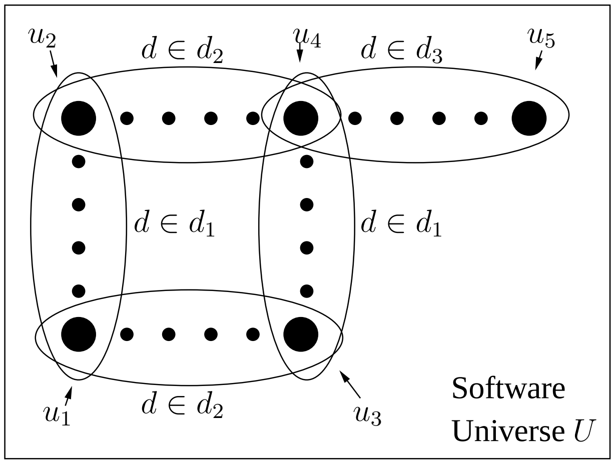
Definition 6 *The composition of diversity techniques can be formally expressed as a path of hyperedges $\mathcal{P}$ on the diversification hypergraph, where two edges are* adjacent *in the path if and only if their intersection contains at least one element of U.*

The act of composing multiple diversity techniques can be thought of in terms of the diversity hypergraph H as moving from one binary in U to another by walking from one adjacent hyperedge to the next. In Figure 3.1, OS diversification and hardware diversification is applied to a single Apache web server installation. By composing these two diversity techniques, the system engineer would force an attacker who is able to directly exploit Apache on a Windows machine to mutate his attack for both a different operating system, namely OpenBSD, and a different hardware platform.

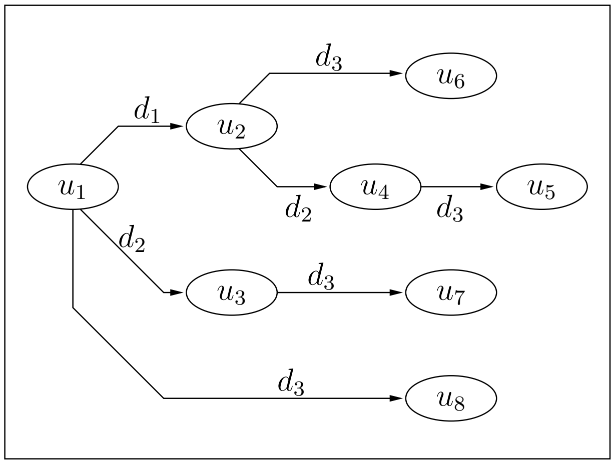
Definition 7 Temporal Precedence *is an ordering on all diversity techniques necessitated by the stage in the design process where the techniques must be applied.*

The application of one diversification technique may undo the work of a previously applied technique. Therefore, two diversity techniques can be composed if and only if they respect *temporal precedence*. A simple but illustrative example of temporal precedence can be found in the use of both source code modification and compile time automatic variable location randomization diversity techniques. Both techniques can be utilized to make a single software package more diverse than its standard, reference compilation. The temporal hierarchy places any source code modification before the address space randomization since it is necessary for any source code modification techniques to be applied before any address space randomization techniques are considered. We deconstruct the temporal hierarchy into the diversification stages discussed in Section 2.1, namely *Requirements*, *Architecture*, *Implementation*, and *Realization*.

Both composition and precedence requirements can be visualized in Figure 3.2(a). In the example, software package u_1 belongs to two equivalence classes generated by diversity techniques d_1 and d_2. The diversity technique d_1 encompasses a large number of diversified software packages, including u_2, which is in turn further diversified by technique d_2. Similarly, package u_3 is related to u_1 by diversity technique d_2, and is then further diversified by d_1. The diversity techniques d_1 and d_2 create hyperedges which form a path from u_1 to



(a)



(b)

Figure 3.2: Figures (a) and (b) provide abstract views of the interaction of diversity techniques. Figure (a) graphically shows the generalized view of software diversity described in Section 3.2, where diverse software instances are set elements, diversification techniques are equivalence classes, and the composition of multiple diversity techniques forms a path across equivalence classes. We represent the a simplified view of the diversification hypergraph H in (b), where the edges represent individual hyperedges and the vertices represent software packages in U which lie at the intersection of two hyperedges.

u_4 through both u_2 and u_3. Since d_1 and d_2 can be applied in any order without violating temporal precedence, the application of d_2 after d_1 to u_1 reaches the same software instance as the application of d_1 after d_2. Finally, we show a single instance of the application of d_3 to u_4, which is 3 diversification techniques away from u_1. While d_3 can be applied to u_1, u_2, and u_3, the equivalence classes created by such an application are omitted from the diagram for the sake of clarity.

3.2.2 Attack and Defense Modeling

The utility of dividing diversified software packages into equivalence classes is more clear when examined through the lens of *attack modeling*. The deployment of a wide variety of commercial-off-the-shelf operating systems to a network may be an effective method of combating a worm which is designed to attack a single exploit, but is ineffective against an attacker who is willing to purchase each of the operating systems and invest the necessary time required to develop a set of custom exploits against each OS. Conversely, a compile time randomization which alters the structure of a binary for each system would be an effective method of combating a human being who develops their exploits using a debugger and a local copy of the software under attack, but would only delay a worm which uses a search algorithm to determine the memory locations of the previously used attack targets.

The diversity schemes discussed are also not equally effective against all forms of attack. Diversifying the instruction sets utilized by different binaries can combat buffer overflow attacks, but the technique is ineffective against a resource exhaustion attack. Producing several versions of the software to utilize different network protocols may evade a denial of service attack yet produce binaries which are vulnerable to a buffer overflow attack.

Definition 8 *Let the types of attacks that would take place be denoted by the set T. We denote the set of software attacks as A. The mapping of attacks on software packages to the attack techniques used is defined as $\tau\colon A \mapsto T$.*

For every diversity technique in D there exists a series of hyperedges in which the vulnerable software package resides. The attack technique $\tau(a)$ employed by an attack $a \in A$ can be mutated to attack another software package which resides in one of the vulnerable software package's equivalence classes. The system designer can then choose which attacks are the most threatening to system survivability by weighting the range of τ to the most critical attack types.

Let M be the set of all implementation metrics which are of interest to the system designer. The implementation metrics can be exhibited in several forms, such as a slowdown associated with the execution of a binary which underwent modification by a diversity technique. In a similar fashion, the increase in runtime memory consumption and program storage size of the diversified binary are also accounted for this way. M is not limited to system performance metrics, however, as the total economic cost incurred by the implementation of diversification techniques can be included in this set.

Definition 9 *The* diversification cost κ *is a function which maps each diversification technique in D and performance metric in M along with the type of binary which is being diversified in U to a positive and real multiplicative factor corresponding to the implementation cost:*

$$\kappa\colon D \times M \times U \mapsto R^+$$

Definition 10 *The* effectiveness probability ρ *is a function which maps each diversification technique in D and attack technique in $\tau(A)$ to a quantity which reflects the ability of a diversification technique to resist the specified form of attack.*

$$\rho\colon D \times \tau(A) \times U \mapsto [0, 1]$$

Each diversity technique has two metrics with which it is associated. The *diversification cost* κ is a function which quantifies the cost to each system performance metric associated with implementing each diversity technique, be it memory consumption, loss of execution speed, or economic cost of implementation. The *effectiveness probability* ρ is a function which quantifies the probability that an attack technique can be modified to compensate for the diversity introduced by a given technique. The effectiveness probability reflects an attacker's ability to mutate an attack against any one binary in the equivalence class to be effective against any other binary in the equivalence class, and is a function of the attacker's skill and the type of attack that is being combated. Metrics of this type have been employed for describing code obfuscation techniques to combat reverse engineering [15].

The effectiveness probability need not be defined for the entire set of attack techniques, as indicated by the choice of $\tau(A)$ rather than $T(A)$. The system designer can choose a subset of attack techniques which he or she considers to be of the greatest threat and model the effectiveness of each diversity techniques against only the attack techniques of interest. Additionally, it is possible to use an element in u rather than the chain of all diversity techniques to define the cost and effectiveness of applying a single diversity technique even though the effectiveness of a diversity technique may be a function of previously applied techniques. Each element in u, by its nature, encodes the set of all diversity techniques which have been previously applied in order to reach that point. The concept of a diversity technique's cost and effectiveness being a function of previously applied diversity techniques is discussed in the following definition.

Definition 11 *The property of* diversity non-linearity *dictates that the cost and effectiveness of a diversity technique is a function of the previously applied diversity techniques. The cost and effectiveness of the currently applied diversity technique can be* amplified *or* attenuated*, which we term a* non-linear composition*. If the cost and effectiveness of a diversity technique is unaffected by previous diversity techniques, we define the interaction as being a* linear composition*.*

The effectiveness and cost of applying a diversity technique to a binary is not constant for all systems. A diversification technique that depends upon linking to functionally equivalent but different standard libraries may cost more to apply for a closed-source operating system than for an open-source operating system. Address space randomization techniques become more effective as the address space available on the hardware platform increases. This property of *non-linear composition* holds implications for the development of algorithms for the optimization of diversity, as shown in Section 3.3.

Definition 12 *We define an* attack relevance *function* $w_\rho \colon \tau(A) \mapsto [0..1]$ *which sets the relative importance of individual attack threats to the system designer. A similar weighting function, or the* cost relevance $w_\kappa \colon M \mapsto [0..1]$, *is provided to balance out the diversification cost.*

A system engineer can then form an *attack model* in which diversity is involved by choosing an appropriate attacker profile and use historical data to generate the effectiveness probability expected for the diversity techniques against the attacker. Furthermore, the system engineer can create a *defense model* consisting of the set of attacks which become critical for system defense. A first attempt at generating a survey of diversity techniques which examines their effectiveness against various classes of attacks is presented in [47]. Both of these functions are utilized in the optimization of diversity assignments, as demonstrated in the following section.

3.3 Hyperpaths and Choosing Diversity

The hypergraph framework presented in Section 3.2 provides a method for determining when and how to apply diversity techniques to a piece of software on a single server and for an entire network of systems. For a single piece of software, the system designer is faced with determining a walk on the hypergraph which provides the greatest distance

Table 3.1: A list of the symbols and functions used in the system model described in this chapter.

Notation:

Primitives			**Functions**		
U	-	Set of all binaries	τ	-	Mapping from attacks to attack techniques
D	-	Set of all diversity techniques			
$H = (U, \mathcal{E})$	-	Diversity hypergraph formed by U and D	κ	-	Cost of implementation of a diversity technique as a function of the diversity technique, cost metric, and the software being diversified.
A	-	Set of attack techniques			
T	-	Types of attacks			
M	-	Non-diversity performance metrics i.e. memory usage, economic cost	ρ	-	Probability of an attack against a diversified software package as a function of the diversity technique, the type of attack, and the software being diversified.
Bounds and Weights					
w_κ	-	Weights the relative importance of different cost metrics			
w_ρ	-	Weights the relative importance of resisting different attack forms			
κ'	-	Bound on the acceptable cost of diversification			

between two diverse software packages while keeping the project under pre-specified cost bounds. When faced with a network of systems, the designer must determine a set of diverse software packages which, when assigned to systems on the network, span the largest distance in the diversification hypergraph if they are neighbors of each other on the network.

In both general cases, we show that determining optimal solutions to both of these problems is NP Hard. For all current practical instances of the host-based diversity assignment problem which can be currently envisioned, however, heuristic methods can be used to determine the optimal choice of diversity techniques. The same is not true for the network diversity assignment problem, as we see in Chapter 4.

3.3.1 Host-Centric Diversity Assignments

For a single server, we are interested in maximizing the workload for an attacker who wants to mutate a commonly available exploit to be effective against our software installation. If we consider the hypergraph generated by our available diversity techniques, we can maximize the attacker workload by finding a path of hyperedges whose total effectiveness probability is minimized and whose total cost is below the cost bound. In terms of diversity techniques, we need to determine the subset of techniques which should be applied to a single host that would maximize the resistance of a host to a known form of attack, given a maximum acceptable performance hit associated with the subset of diversity techniques used. Generating an optimal solution to this problem is NP-Hard, due to its equivalence to the classic bin packing problem [5]. We provide a proof of the problems complexity along with an adaptation of a classic greedy algorithm for generating feasible solutions to the problem in Appendix A.

The state-of-the-art in software diversification does not currently necessitate the use of an NP solver for generating optimal host-level software diversity allocations. Practically, the costs of diversity techniques fall into several narrow bands, ranging from near zero for compiler-driven randomization schemes to the large expense associated with the purchase of multiple hardware platforms or developers to generate diverse code architectures. The wide disparity in costs associated with each technique makes algorithms for determining host-based diversity assignments unnecessary. As more diversity schemes are explicitly characterized by their cost, effectiveness, and temporal precedence, it may become necessary to develop heuristic algorithms for determining optimal diversity in a reasonable amount of time.

3.3.2 Network-Centric Diversity Assignments

A far more interesting problem deals with the assignment of diverse software packages to a network of systems, as first discussed in Section 2.2.1. Unlike host-centric diversity assignments, a network-centric diversity assignment does not necessarily require that every host is maximally different from every other host on the network. The goal of this particular diversity assignment is to increase the difficulty of starting from a single host and leapfrogging from one system on the network to the next. In the remainder of this dissertation, we describe and examine methods of generating network diversity assignments which combat multiple forms of attackers.

Chapter 4. Distributed Diversity Assignment Algorithms

Behold, the people is one, and they have all one language; and this they begin to do: and now nothing will be restrained from them, which they have imagined to do. Go to, let us go down, and there confound their language, that they may not understand one another's speech.

The Book of Genesis

4.1 Network-Centric Diversity Assignments

The network-centric diversity problems deal with assigning diverse software systems to specific hosts on a network topology in order to increase the difficulty associated with attacking the network. More specifically, we want to determine an assignment of diverse software packages to combat two different attack models, both presented in Table 4.1. The first model describes a network administrator who faces an intelligent adversary that is able to build new attacks, recompile previously designed attacks for new platforms, and learn from previous experiences. The second model attempts to encapsulate the behavior of a computer virus which can rapidly propagate to all neighboring systems which are of similar design.

When compared to the host-centric diversity problem, the assignment of diverse software packages to hosts on a network of systems is far less trivial. The problem requires the generation of an optimal host-centric diversity assignment for every pair of adjacent hosts on a defined network topology; as we see later, this problem is NP-Hard for all but the simplest of network topologies.

Before moving forward, we need to state the network diversity assignment problem in terms of the formalism provided in Chapter 3. The function $k\colon V \mapsto U$ maps a software package to a host on a network. The network topology is represented by the graph $G = (V, E)$. $T' \subseteq T$ is the set of attack techniques which can lead to a host compromise that

Table 4.1: A comparison of the Intelligent Attacker vs. Diverse Network and Computer Virus vs. Diverse Network attack and defense models.

	Intelligent Attacker	Diverse Network
Attack Model	We assume that an intelligent attacker holds a set of exploits which are specific to the target platform and the launch platform. In their basic form, the exploits are built to be launched from the same platform as their target. The attacker can modify the exploit to be launched from new platforms as well as modify the exploit to be effective against new platforms. Furthermore, the attacker can use a compromised node to attack newly adjacent nodes.	We assume that a virus can take over a node and use the node to further propagate the attack. It cannot modify its attack to compromise targets which are dramatically different from the host system.
Cost of Attack or Effectiveness of Diversity	If two adjacent systems are running a similar software package and it is the first time the attacker is being exposed to the pairing, the probability of an attack propagating from one host to another is a function of the number of diversity techniques used to separate the two software packages. If an attacker has been previously exposed to the pairing, then compromising the target node from the source node is of minimal cost to the attacker.	If two adjacent systems are running a similar software package, the probability of an attack propagating from one host to another is a function of the number of diversity techniques used to separate the two software packages.
Defense Model	We want to slow an attacker from taking over the entire network by leapfrogging from one compromised host to the next by presenting the attacker with as many different launch-platform / target-platform pairings as possible.	We want to prevent a virus with a small set of attacks from taking over the entire network by leapfrogging from one compromised host to the next.
Composition	The composition of a set diversity techniques increases the attack resistance of a software package by the product of the *effectiveness probability* metrics of each of the diversity techniques.	The composition of a set diversity techniques increases the attack resistance of a software package by the product of the *effectiveness probability* metrics of each of the diversity techniques.

an attacker can use to further propagate the attack. As defined in Chapter 3, we define the effectiveness probability of each diversity technique be ρ. $\mathcal{P}'_{max}(v_s, v_t) \subseteq \mathbf{P}'(v_s, v_t)$ is the path of maximum attack similarity, as defined by ρ, which connects two applications v_s and v_t covered by the hypergraph H. Let s be a function which computes the similarity between two binaries:

$$s(t) = \begin{cases} 1 & : \quad k(v_s) = k(v_t) \\ \prod_{d \in \mathcal{P}'_{max}(v_s, v_t)} \rho(d, t) & : \quad \text{else} \end{cases}$$

The network topology represented by the graph G is by no means a complete graph. While full connectivity is usually assumed for random scanning Internet worms [13], the formulation and algorithms presented in this dissertation are general enough to encompass attacks which propagate across non-complete topologies, which are common in the computing space [51]. This is critical for combating worms that leverage file sharing networks [43] or email communication [40, 41] for transmission.

We can formally define two versions of the network diversity assignment problem, referred to as the INTELLIGENT ADVERSARY NETWORK DIVERSITY ASSIGNMENT and the AUTOMATED ADVERSARY NETWORK DIVERSITY ASSIGNMENT problems. In the first of the two problems, our goal is to create a diversity assignment where an attacker who is traversing the network is presented with novel edges on each step in their path. The second of the two problems relaxes the constraint requiring edge novelty and attempts to reduce the number of adjacent nodes which run similar software packages.

4.1.1 Slowing Intelligent Attackers

As stated, an optimal solution to the intelligent adversary problem would be a diversity assignment which presents the attacker with novel pairings of launch platforms and target platforms when the adversary traverses the network. We must first define two components of the optimization measure.

Definition 13 *A non-repetitive walk is a walk on a graph which does not traverse any vertex twice.*

Definition 14 *An edge color pair is an ordered pair created by the color of the head and the tail of an edge.*

Using these two terms, we can more precisely state the measure of the optimization problem as follows:

INTELLIGENT ADVERSARY NETWORK DIVERSITY ASSIGNMENT:

- INSTANCE: Graph $G = (V, E)$, a hypergraph H, a set U, and a set $T' \subseteq T$.
- SOLUTION: An instance $k \colon V \mapsto U$
- MEASURE: The number of unique edge color pairs present on every non-repetitive walk in G given that $\forall \{v_s, v_t\} \in E \colon k(v_s) \neq k(v_t)$

We show in the following theorem that maximizing the number of unique, ordered color pairs on every non-repetitive walk in G is NP-Hard for general graphs. This is done through a reduction to the harmonious coloring problem [53].

Definition 15 *A harmonious coloring of a graph is a proper vertex coloring where every edge color pair present on the graph is unique.*

In a harmonious coloring, a proper vertex coloring of a graph is generated and the endpoint colors of every edge form an unordered pair which colors the edge. The graph is harmoniously colored if and only if no two edges possess the same unordered color pair.

Theorem 1 *The* INTELLIGENT ADVERSARY NETWORK DIVERSITY ASSIGNMENT *optimization problem is NP-Hard.*

Proof: Consider an undirected graph G that contains an Euler tour and a set of colors in U, where $|U| < |V|$. Convert G to a directed graph G', where every edge in G is a pair of bidirected edges in G'. If an algorithm can generate a feasible solution for the INTELLIGENT ADVERSARY NETWORK DIVERSITY ASSIGNMENT problem on G', then the corresponding vertex coloring on G is a harmonious coloring. ■

While the INTELLIGENT ADVERSARY NETWORK DIVERSITY ASSIGNMENT problem is not solvable in P assuming that P $\neq$ NP for general graphs, it is solvable in linear time for trees. We can label a subset of nodes as being critical to the network's operation, then generate a derived graph which contains a node that is the union of all the critical nodes. The node which is generated by union of the critical nodes then becomes the root of a Breadth First Search (BFS) tree. Each of the nodes located in the BFS tree are then colored so that any non-repetitive walk on the tree generates a deBruijn sequence. The resultant coloring forces all edges on the graph which lead to vertices that are closer to the root nodes to be "challenging", or novel to the attacker.

In Figure 4.1, we present a distributed BFS algorithm for solving the intelligent adversary problem. All nodes are initialized with a random software package. The algorithm initiates a depth counter at the root nodes, which then transmit the counter to their neighbors. All nodes are instructed to retransmit an incremented depth counter upon reception of a depth which is smaller than their currently held value. The depth value is used as an index for a *deBruijn sequence* [38], which a sequence of symbols which do not contain subsequence repetitions of a fixed-sized. We generate our order 2 deBruijn sequence by computing an Euler tour on a complete bidirected graph, where the nodes are labeled by the software packages available to each node.

In Figure 4.2, we compare the BFS-based algorithm to a purely randomized assignment of diverse software packages on a tree. For sparse, connected graphs of small diameter with a large number of available software packages, the randomized software assignment per-

```
# $U := [U_1, \ldots, U_k]$ is the set of software packages
deBruijnSequence():
    sequence := Null
    euleredges := Null
    for head $:= 1$, head $<= |U| - 1$, head$++$:
        for tail $:=$head$+1$, tail $<= |U|$, head$++$:
            if euleredges := Null:
                euleredges.append((i,j), (j,i))
            else:
                circuit := (i,j) (j,i)
                Splice circuit into euleredges
    for edge in euleredges:
        sequence.append(head(edge))
    return sequence

For each node:
    if not a parent node:
        ParentNode := Null
        Depth := Null
    else:
        Depth := 0
        ParentNode := Self
        deBruijn := deBruijnSequence()
    CurrentColor := Randomly chosen value from $U$
    Initialize CurrentColor
    Sleep for an exponentially distributed random period
        and execute EventLoop() on wakeup.

EventLoop():
    if not a parent node and a depth update was received while asleep:
        if Depth := Null or ReceivedDepth $<$ Depth:
            Depth := ReceivedDepth
            ParentNode := Node ID of node which transmitted our new depth
        Set CurrentColor to $U[deBruijn[Depth \% |U| * |U - 1|]]$
        Initialize CurrentColor
    Transmit Depth + 1 to neighbors
    Transmit my node ID to neighbors
    Sleep for an exponentially distributed random period.
```

Figure 4.1: BFS Based Algorithm for solving a restricted version of the INTELLIGENT ADVERSARY DIVERSITY ASSIGNMENT PROBLEM

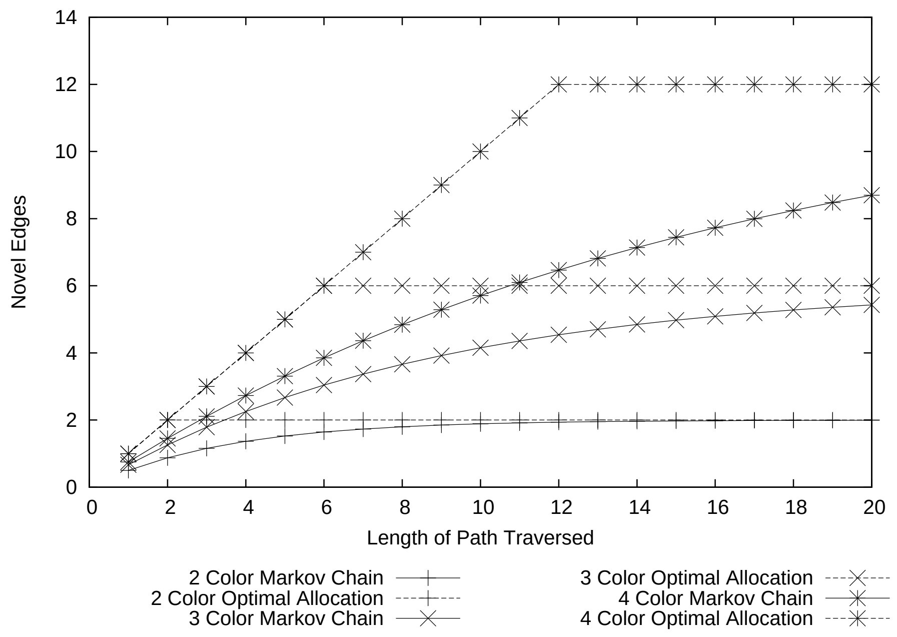


Figure 4.2: Plot of the number of novel edges vs. the length of the traversed path. For small diameter networks with a large number of colors, or large diameter networks with a small number of colors, it is unnecessary to use the BFS algorithm.

forms as well as the BFS-based algorithm. The combination of software set size and small diameter prevents the set of novel edges from being completely exhausted before a repeated edge is generated. Similarly, for networks with a large diameter and small software set, the number of edges eventually exhausts all possible novel edges. Dense graphs will increase the number of parallel paths an attacker can traverse, thus decreasing the performance of the randomized algorithm. The BFS-based algorithm will be unaffected, since the shortest path from any node to the critical nodes is guaranteed to have a correct software assignment.

It is trivial for an attacker to prevent the BFS-based algorithm from performing correctly. After compromising a single node at the edge of the network, an attacker can force the compromised node to report itself as being a root node. The neighboring nodes will accept the new depth update, and correspondingly change their software package. The adversary can continue to push his or her attack forward by compromising nodes and then fooling their neighbors to choose a different color. Rather than having to traverse a multitude of novel edges, the attacker will be able to take a single edge node, push a desired coloring onto neighboring nodes, compromise a new neighbor, then repeat the process until he or she reaches a critical node.

Optimally, we would like to combat any algorithm-driven attack using the principles of diversity, where *a diversity of algorithms for assigning diversity are used to slow or stop an attacker*. In the following section, we show how this can be done.

4.1.2 Stopping Viruses and Limited-Skill Attackers

The vast majority of intrusions experienced by computer systems are not waged by intelligent adversaries. More commonly, systems are penetrated either by a virus that repetitively utilizes a single attack or by an unskilled attacker who applies an unmodified, pre-written exploit to attack machines of a single system type. We want to define a new diversity assignment goal, one which reduces the number of times globally that an attacker can leapfrog from one identical system type to the next, while isolating as many nodes of the same system type from one another. Therefore, our optimization objectives become:

1. A minimization of the number of neighbors running the same software packages
2. A maximization of the number of disconnected “islands” of nodes running the same software packages

These objectives, referred to as the defective edge count and the connected component count, are not orthogonal. A local reduction in the number of neighbors running the same software package globally reduces the number of edges an attacker can use to propagate an attack. A global increase in the number of disconnected components increases the number of initial nodes that must be taken by an attacker if he or she wishes to compromise every node on the network. If there are no neighbors running the same software package, then every node is a disconnected “island”. More formally, the problem of combating automated adversaries using diversity assignments can be stated as follows:

AUTOMATED ADVERSARY NETWORK DIVERSITY ASSIGNMENT:

- INSTANCE: Graph $G = (V, E)$, a hypergraph H, a set U, a set $T' \subseteq T$, and a similarity function s.
- SOLUTION: An assignment instance $k\colon V \mapsto U$
- MEASURE: The total probability of effectiveness of attack across all node pairs:

$$\sum_{\{v_s, v_t\} \in E} s(v_s, v_t, T')$$

In general, however, this problem is also NP-Hard:

Theorem 2 *The* AUTOMATED ADVERSARY NETWORK DIVERSITY ASSIGNMENT *optimization problem is NP-Hard.*

Proof: Consider an assignment of diverse software package where only one diversity technique exists. The range of s is therefore limited to two discrete values, $\{0, 1\}$. In order to minimize the total value of s across all links in the graph, the number of links which connect diverse systems must be maximized. Any algorithm which can assign diverse software packages to nodes in a network so that the number of edges between diverse software packages is maximized would also be able to solve MAX-K-CUT. Therefore, any algorithm which can solve the NETWORK DIVERSITY ASSIGNMENT problem in polynomial time would be able to do the same for MAX-K-CUT. ■

If we consider s to be a discrete function, as described in the above proof, the current network diversity assignment problem is similar to another classic graph theory problem. The assignment of the software packages in U to the graph G is what graph theoreticians would call a *coloring* of graph G. The assignment of colors in such a way that the number of *defective* edges, or communication links that exist between two nodes of the same color, is minimized is called an *optimum coloring*. A *perfect coloring* is an assignment of the minimum number of colors necessary to color a graph such that no two neighboring nodes share the same color. The minimum number of colors required for a perfect coloring is denoted by $\chi(G)$. When $|U| < \chi(G)$, any color assignment will induce at least one edge where both endpoints are similarly colored. A coloring where such an edge, referred to as a defective edge, is present is called a *defective coloring*.

We use the terms *colors* and *software packages* interchangeably throughout the rest of the dissertation.

Determining a minimum number of colors required to achieve a perfect coloring is, in the general case, an NP-Hard problem [5]. Aside from a handful of special cases, determining an optimum coloring with a minimum number of defective edges is also NP-Hard [21].

In the remainder of the chapter, we provide a class of algorithms which assigns software packages to nodes on a communication network in order to limit the total number of nodes an attacker can compromise using a limited attack toolkit. Our algorithms are based on examining local information and making local decisions. They work by directly decreasing the defective edge count and indirectly improving the connected component count. We have examined these algorithms through analysis and simulation on real-world graphs, as shown in Sections 4.2 and 4.3.1.

Given the purpose of the software distribution algorithm, it is logical to explore the vulnerability of the coloring algorithms themselves from the standpoint of an attacker. Based upon this reasoning, we have developed a series of attacks against our own algorithms and explored their effectiveness through simulation. These attacks do not rely upon attacking implementation flaws in the algorithms, but instead are based on malicious nodes attempting to deceive well-behaving nodes running the algorithm. The results of this simulation work are presented in Section 5.2.1.

In Section 5.2.2, we draw several conclusions from our examination of the simulation results. Our explorations of the attacks' effects on the coloring algorithms presented give rise to the observation that *there exists a tradeoff between an algorithm's tolerance to attack and the quality of the software assignment created by the algorithm.* Furthermore, we show that revisiting the initial thesis on the value of diversity is applicable in the design of software assignment scheme when an algorithm designer wishes to increase the algorithm's tolerance to a directed attack. More precisely stated, we conclude that *diversity must be introduced at all levels of the system design, including any scheme that is used to introduce diversity itself.*

4.1.3 Related Work

Inspiration for the examination of a network from the standpoint of an attacker's progress in conquering multiple connected computer systems is drawn from attack graph research [69]. In general, an attack graph is a graph theoretic representation of an attacker's ability to attain attack states, represented by nodes, and the techniques used to attain those states, represented by edges. Much of this research has concentrated on efficient ways of generating these graphs [3, 45]. Suggestions on how to improve the security of an attack graph relies upon having absolute knowledge of vulnerabilities on each node.

The similarities between the topological properties of human social relations and the Internet allow us to examine research originally intended for preventing human epidemics in the context of computer hackers and viruses [27, 28, 62, 66, 67]. It has been shown that in certain classes of network topologies, any infection, under standard models, would become an epidemic. Additionally, they state that an epidemic can be stopped by conducting selective immunization of nodes based on their node degree. High-degree nodes are essential for the connectivity of the network, and removing even a small fraction of them can quickly disconnect the graph [1]. While it would be possible to install different software based solely upon node degree, unequal protection against an attack would occur. A worm that would attack the software population's low-degree nodes would have difficulty in spreading and would not compromise the network. An attack against the software assigned to the high-degree nodes would be able to rapidly propagate and disconnect the network.

4.2 Distributed Algorithms

As stated previously, we have designed and analyzed a series of distributed algorithms which seek to minimize the number of defective edges present on a communication graph. The algorithms are presented in order of increasing complexity of implementation. The RANDOMIZED COLORING algorithm presented in Section 4.2.1 requires each node to randomly select its color and not change it throughout the duration of the network's operation. The second algorithm allows a node, at random intervals, to examine its local neighborhood and choose a new color for itself if a large number of its neighbors have the same color. We refer to this algorithm as the COLOR FLIPPING algorithm, and it is presented in Section 4.2.2. The next pair of algorithms, referred to as the COLOR SWAPPING algorithms, allows pairs of nodes, again at random intervals, to swap their colors in order to reduce the number of defective edges. These are presented in Section 4.2.3. Finally, a pair of algorithms which combine both color flipping and color swapping strategies are presented in Section 4.2.4.

Each of these algorithms is presented alongside their implementations in pseudocode. Functions which are common to each of the algorithms are presented in separate listings. For example, all of the algorithms presented rely upon each maintaining a set of local variables, such as the set of colors available, basic tools for querying the status of a neighbor's coloring, and an event loop. The pseudocode for these components is presented in Figure 4.3.

```
Define $|U|$ as number of available colors
Define $U$ as set of all available colors
Define NeighborCount as number of neighbors
Define NeighborSet as set of all neighbors
Define CurrentColor for each node

EventLoop():
    if the timer event has occured:
        DoRecoloring()
        Set new timer event
    Continue EventLoop()

ComputeDefect():
    for each Neighbor in NeighborSet:
        ColorQuery(Neighbor)
    return number of nodes running CurrentColor

ColorQuery(Neighbor):
    Query Neighbor for its current color and store in NeighborColor
    return NeighborColor

SwapImprovementQuery(Neighbor):
    Query Neighbor for its improvement in defective edge count if
        Neighbor executes ComputeSwappedDefect(Self)
    Store Neighbor's response in NeighborDefect
    return NeighborDefect

DoSwapQuery(Neighbor, NeighborColor):
    Instruct Neighbor to do a color swap
    if Neighbor denies request:
        return AbortedSwap
    else:
        CurrentColor := NeighborColor
        Initialize CurrentColor
        return SwapComplete
```

Figure 4.3: Pseudocode for various globally shared functions and variables used by the distributed coloring algorithms.

```
RandomizedColoring():
    CurrentColor := a random package from $U$
    Initialize CurrentColor
```

Figure 4.4: Pseudocode implementation of the Randomized Coloring algorithm.

4.2.1 Randomized Coloring

The first, and most basic, algorithm discussed is the RANDOMIZED COLORING algorithm, as shown in Figure 4.4. This provides, on average, $m/|U|$ defective edges. Proving this is a simple exercise: after randomly coloring every node on the graph, select a single edge. The probability that both endpoints have the same color is $1/|U|$. Summing across all edges, the average number of defective edges is $m/|U|$. The algorithm requires $O(1)$ time to run on each node, and zero communication between the nodes is required. Because of the lack of inter-node communication, the algorithm can be considered extremely secure against attack.

The graph coloring provided by the algorithm, however, is sub-optimal. In the worst case, this algorithm performs poorly. A randomized algorithm may lead to every link forming a connection between two identical systems. While the probability of this event occurring is $(1/|U|)^{n-1}$, the result would have a significant impact on system security.

4.2.2 Color Flipping Algorithms

In the COLOR FLIPPING algorithm shown in 4.5, nodes initialize themselves by executing the randomized coloring presented in Section 4.2.1. After a random delay, each node performs a local search amongst its immediate neighbors to determine if switching to a new color would decrease the number of locally defective edges. Since each node must now poll its immediate neighbors to discover their current color, the algorithm requires $O(\Delta(G))$ time to poll the neighbors per cycle, where $\Delta(G)$ is the maximum degree of the graph. After the data is collected, $O(\Delta(G) + |U|)$ operations must be done to generate a census of the local colors and determine the minority color.

If it is discovered that switching to the minority color would decrease the local defect to below $d(v)/|U|$, then the flip is instantiated. It can be easily shown that the COLOR FLIPPING algorithm will converge. Each color flip reduces the number of defective edges by at least 1. The number of edges present in the graph is m. The maximum number of color flips that can therefore be conducted is m. Similar proofs can be found throughout the literature; Vazirani leaves the proof as an exercise to the reader in [84]. By the time the algorithm has converged, total number of defective edges is provably decreased below the average number of defects in the RANDOMIZED COLORING algorithm:

Theorem 3 *The upper bound on the number of defective edges produced by* COLOR FLIPPING *is no more than the average number of defective edges produced by* RANDOMIZED COLORING.

Proof: At the point of convergence, each node is connected to at most $\lfloor d(v)/|U| \rfloor$ defective edges. The number of defective edge endpoints is $\sum_v \lfloor d(v)/|U| \rfloor$. The number of defective edges is therefore $1/2 \sum_v \lfloor d(v)/|U| \rfloor$. In comparison to the randomized algorithm:

$$\frac{1}{2}\sum_v \left\lfloor \frac{d(v)}{|U|} \right\rfloor \leq \frac{1}{2}\sum_v \frac{d(v)}{|U|} = \frac{m}{|U|}$$

■

4.2.3 Color Swapping Algorithms

The following pair of algorithms are extensions of the Kernighan-Lin heuristic [5] for computing balanced cuts. In both algorithms, each node attempts to reduce its number of

```
RespondToColorQuery(Neighbor):
    Transmit CurrentColor to Neighbor

ComputeMinorityColor():
    ColorCount[0..$|U|-1$] := 0
    for each Neighbor:
        ColorCount[ColorQuery(Neighbor)]$++$
    MinorityColor := 0
    for $i = 1..|U|-1$:
        if ColorCount[i] $>$ ColorCount[MinorityColor]:
            MinorityColor := i
    return $\{$MinorityColor, ColorCount[MinorityColor]$\}$

FindBestFlip():
    CurrentDefect := ComputeDefect()
    if CurrentDefect $>$ NeighborCount / ColorCount:
        $\{$ProposedColor, NewDefect$\}$ := ComputeMinorityColor()
        if CurrentDefect - NewDefect $>$ 0:
            return $\{$ProposedColor, CurrentDefect-NewDefect$\}$
    else:
        return NoFlipFound

DoFlip():
    $\{$NewColor, DefectImprovement$\}$ := FindBestFlip()
    if NewColor $\neq$ CurrentColor:
        Initialize CurrentColor

For each node:
    Call RandomizedColoring()
    Set a timer event
    Answer neighbor queries using RespondToColorQuery()
    Redefine DoRecoloring() as DoFlip()
    Start the EventLoop()
```

Figure 4.5: Pseudocode implementation of the Distributed Color Flipping algorithm.

```
ComputeSwappedDefect(SwapPartner):
    ColorCount[0..$|U| - 1$] := 0
    for each Neighbor:
        if Neighbor = SwapPartner:
            ColorCount[CurrentColor]++
        else:
            ColorCount[ColorQuery(Neighbor)]++
    NewDefect := ComputeDefect()
    DefectImprovement := NewDefect - ColorCount[ColorQuery(SwapPartner)]
    return

FindBestSwap():
    MyDefect :=ComputeDefect()
    ExpectedSwapGain := 0
    FoundASwap := FALSE
    for each Neighbor in NeighborSet:
        NeighborColor := ColorQuery(Neighbor)]
        if NeighborColor $\neq$ CurrentColor:
            MySwapGain := ComputeSwappedDefect(Neighbor)
            NeighborSwapGain := SwapImprovementQuery(Neighbor)
            if AcceptablePartner(MySwapGain, NeighborSwapGain):
                ExpectedSwapGain := NeighborSwapGain + MySwapGain
                SwapPartner := Neighbor.
                SwapPartnerColor := NeighborColor
                FoundASwap := TRUE
    if FoundASwap = TRUE:
        return {SwapPartner, SwapPartnerColor}
    else:
        return {FALSE, FALSE}

DoSwap():
    {SwapPartner, SwapPartnerColor} := FindBestSwap()
    if SwapParnter = FALSE:
        return FALSE
    Result := DoSwapQuery(SwapPartner, SwapPartnerColor)
    return Result
```

Figure 4.6: Pseudocode that describes support functions used by the distributed color swapping algorithms.

defective edges by negotiating for a color "swap" between itself and its neighbors. After collecting the number of defective edges which would be removed from the neighbor node and itself by conducting a swap from each neighbor, the initiating node executing the algorithm chooses a neighbor which it views to be optimal and proposes a color swap. If the neighbor agrees to the swap, the initiating node takes the color of the neighbor and the neighbor takes the color of the initiating node. A collection of supporting functions associated with the swap algorithms is presented in Figure 4.6; a separate block of pseudocode which contains the communication functions required for the swap operation is presented in Figure 4.7.

For a swap to take place in the first algorithm, known as MUTUALLY BENEFICIAL SWAPPING and presented in Figure 4.8, the exchange of colors must reduce the defective edge count for both nodes involved. The second algorithm, referred to as GREATER GOOD SWAPPING and presented in Figure 4.9, will incur a swap if the total number of defective edges between both nodes is reduced by the exchange. The greater number of nodes that are available for a GREATER GOOD SWAPPING execution means the quality of the solution associated with the GREATER GOOD SWAPPING algorithm is expected to be better than that associated with the MUTUALLY BENEFICIAL SWAPPING algorithm. Correspondingly, the increased number of swap partners increases the vulnerability of the algorithm to attack. This phenomenon is discussed further in Chapter 5.

4.2.4 Hybrid Algorithms

The final set of algorithms are hybrids of the color swapping and color flipping schemes presented in Sections 4.2.2 and 4.2.3, respectively. The RANDOMIZED HYBRID algorithm, shown in Figure 4.10, requires that a node which wishes to change its color to randomly choose to execute either the GREATER GOOD SWAPPING algorithm or the COLOR FLIPPING algorithm. The selection between the GREATER GOOD SWAPPING algorithm and

```
SwapImprovementQuery(Neighbor):
    Query Neighbor for its improvement in defective edge count if
        Neighbor executes ComputeSwappedDefect(Self)
    Store Neighbor's response in NeighborDefect
    return NeighborDefect

DoSwapQuery(Neighbor, NeighborColor):
    Instruct Neighbor to do a color swap
    if Neighbor denies request:
        return AbortedSwap
    else:
        CurrentColor := NeighborColor
        Initialize CurrentColor
        return SwapComplete

RespondToSwapImprovementQuery(Neighbor):
    MyDefect := ComputeDefect()
    NewDefect := ComputeSwappedDefect(Neighbor)
    DefectImprovement := MyDefect - NewDefect
    Transmit DefectImprovement to Neighbor

RespondToMutuallyBeneficialSwapRequest(Neighbor, NeighborColor)):
    if ComputeSwappedDefect(Neighbor) $\geq 1$:
        Transmit AcceptedRequest to Neighbor
        CurrentColor := NeighborColor
        Initialize CurrentColor
        return SwapComplete
    else:
        Transmit DeniedRequest to Neighbor
        return AbortedSwap

RespondToGreaterGoodSwapRequest(Neighbor, NeighborColor)):
    Transmit AcceptedRequest to Neighbor
    CurrentColor := NeighborColor
    Initialize CurrentColor
    return SwapComplete
```

Figure 4.7: Pseudocode that describes swap query and response functions used by the distributed color swapping algorithms.

```
MutuallyBeneficialPartner(MySwapGain, NeighborSwapGain):
    if MySwapGain $\geq$ 1 and NeighborSwapGain $\geq$ 1:
        return TRUE
    else:
        return FALSE

For each node:
    Do RandomizedSoftware()
    Set a timer event
    Enable RespondToSwapImprovementQuery(Neighbor)
    Enable RespondToMutuallyBeneficialSwapRequest(Neighbor, NeighborColor)
    Redefine AcceptablePartner(MySwapGain, NeighborSwapGain) as
        MutuallyBeneficialPartner(MySwapGain, NeighborSwapGain)
    Redefine DoRecoloring() as DoSwap()
    Start the EventLoop()
```

Figure 4.8: Pseudocode implementation of the Mutually Beneficial Color Swapping algorithm.

the COLOR FLIPPING algorithm does not need to be unbiased; on the contrary, it may be beneficial from a convergence rate or attack tolerance standpoint for the algorithm to prefer one coloring scheme over the other. Determining the optimal point between conducting a flip or a swap can possibly be done through the use of game theoretic analysis, as discussed in Chapter 6.

The BEST CHOICE HYBRID, shown in Figure 4.11 algorithm allows pairs of nodes to examine the defective edge reduction that is possible by either doing a color swap as a pair or independently doing a color flip. If each node in a swap can eliminate a greater number of defective edges by cooperating and performing a swap as compared to individually performing a flip, a swap is conducted. If either of the two nodes finds it can better serve itself by conducting an independent color flip, then a swap is not conducted. If the node

```
GreaterGoodPartner(MySwapGain, NeighborSwapGain):
    if MySwapGain + NeighborSwapGain $\geq 1$
        return TRUE
    else:
        return FALSE

For each node:
    Do RandomizedSoftware()
    Set a timer event
    Enable RespondToSwapImprovementQuery(Neighbor)
    Enable RespondToGreaterGoodSwapRequest(Neighbor, NeighborColor)
    Redefine AcceptablePartner(MySwapGain, NeighborSwapGain) as
        GreaterGoodPartner(MySwapGain, NeighborSwapGain)
    Redefine DoRecoloring() as DoSwap()
    Start the EventLoop()
```

Figure 4.9: Pseudocode implementation of the Greater Good Color Swapping algorithm.

that initiates the re-coloring attempt finds that a swap is not feasible, it attempts to conduct an independent color flip.

4.3 Simulation

In order to test our algorithms, it was necessary to acquire a topology that is representative of the networks that our distributed coloring algorithm would expect to encounter. As many researchers consider generation of a simulated, representative network topology to be an open research problem [59, 93], we have decided to capture an *actual* topology for our algorithm simulation.

For our simulation experiments, we examine a topology generated by e-mail traffic inside the ECE Department at Drexel University. We captured a sample of the logs created

```
**FlipOrSwap()**:
    *FlipOrSwapProbability := .5*
    **if** *UniformRandomNumber[0..1]* $<$ *FlipOrSwapProbability*:
        **DoFlip()**
    **else**:
        **DoSwap()**

**For each node**:
    Do **RandomizedSoftware()**
    Set a timer event
    Enable **RespondToSwapImprovementQuery(***Neighbor***)**
    Enable **RespondToGreaterGoodSwapRequest(***Neighbor, NeighborColor***)**
    Redefine **AcceptablePartner(***MySwapGain, NeighborSwapGain***)** as
        **GreaterGoodParter(***MySwapGain, NeighborSwapGain***)**
    Redefine **DoRecoloring()** as **FlipOrSwap()**
    Start the **EventLoop()**
```

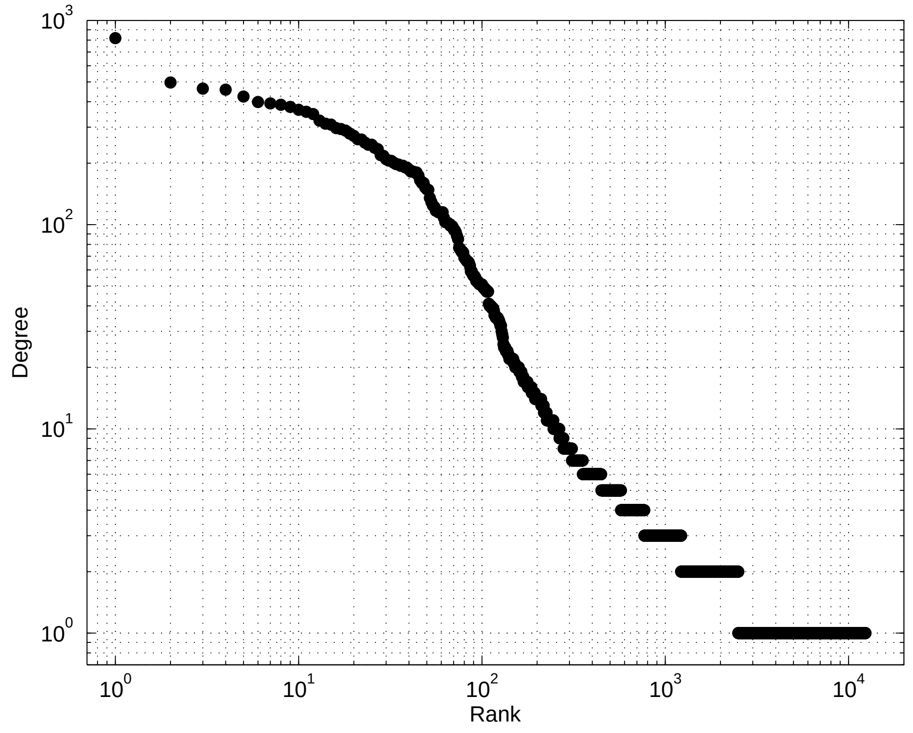
Figure 4.10: Pseudocode implementation of the Randomized Hybrid coloring algorithm.

by e-mails as they passed through the `ece.drexel.edu` server. The raw data consisted of $1,038,939$ log entries for each e-mail sent and received by $278,435$ unique accounts handled by `ece.drexel.edu`'s `sendmail` server from January 13th to September 19th of 2003. Of the original $1,038,939$ e-mails recorded, there are $337,532$ unique $\{$to, from$\}$ e-mail address pairs. This means, strictly according to the logs, there are $337,532$ unique pairs of individuals using the mail server to communicate.

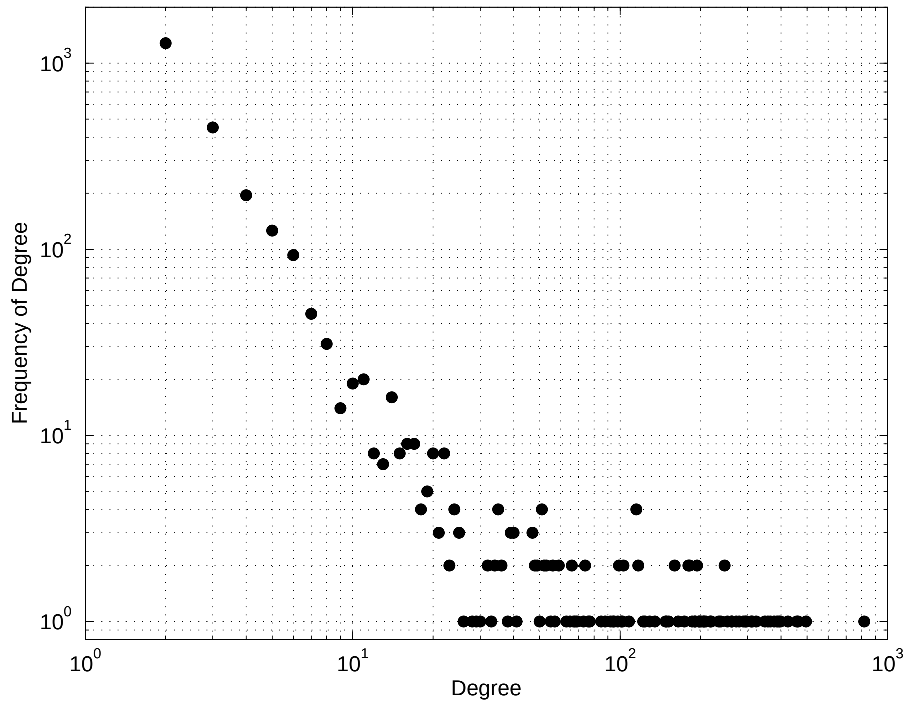
To reduce the impact of spam on our data set, we preserve those edges where, for each sender and receiver, at least one e-mail is sent from the initial message receiver to the initial message sender. This represents a complete communication between the two e-mail entities. Our data set is then reduced to $37,618$ $\{$to, from$\}$ address pairs, or $18,809$ undirected edges. These edges exist between $12,408$ nodes, or unique e-mail ID's, in 14

```
BestChoiceSwapResponse(Neighbor, NeighborColor):
    {FlipColor, FlipDefectImprovement} := FindBestFlip()
    SwapDefect := ComputeSwappedDefect(Neighbor)
    if SwapDefect > FlipDefect:
        Transmit AcceptedRequest to Neighbor
        CurrentColor := NeighborColor
        Initialize CurrentColor
        return SwapComplete
    else:
        Transmit DeniedRequest to Neighbor
        return AbortedSwap

DoBestChoiceHybrid():
    {FlipColor, FlipDefectImprovement} := FindBestFlip()
    {SwapPartner, SwapPartnerColor} := FindBestSwap()
    SwapDefect := ComputeSwappedDefect(Neighbor)
    if SwapParnter = TRUE and SwapDefect > FlipDefectImprovement:
        Result := DoSwapQuery(SwapPartner, SwapPartnerColor)
        if Result = AbortedSwap:
            DoFlip()
    else:
        DoFlip()

For each node:
    Do RandomizedSoftware()
    Set a timer event
    Enable RespondToSwapImprovementQuery(Neighbor)
    Enable BestChoiceSwapResponse(Neighbor, NeighborColor)
    Redefine AcceptablePartner(MySwapGain, NeighborSwapGain) as
        GreaterGoodPartner(MySwapGain, NeighborSwapGain)
    Redefine DoRecoloring() as DoBestChoiceHybrid()
    Start the EventLoop()
```

Figure 4.11: Pseudocode implementation of the Best Choice Hybrid coloring algorithm.



(a)



(b)

Figure 4.12: Log-Log Plots of E-Mail Graph Statistics. The properties of the collected data are statistically similar to many other topologies, including the AS topology seen in BGP routing.

separate connected components, where the largest connected component consists of $12,354$ nodes and $18,768$ undirected edges. Our simulation studies use this largest connected component.

It is customary in the study of large-scale network topologies to examine the distribution of node degrees on a log-log plot. Accordingly, we have plotted the degree of each node versus its rank in a sorted list along with the frequency of degree versus the degree of the node. These plots, whose distribution is consistent with the work of [28, 62], are shown in Figures 4.12(a) and 4.12(b), respectively.

4.3.1 Algorithm Simulation

The coloring algorithms presented in Sections 4.2.1, 4.2.2, 4.2.3, and 4.2.4 are provided with three distinct colors, and are each executed by the $12,354$ nodes at intervals determined by a Poisson process running at each node. The Poisson rate λ is set to $1/n$ algorithm executions per cycle for each node in order to normalize the execution rate of the algorithm by each node with respect to graphs that differ in node count, allowing for an unbiased comparison of the algorithm's performance across varying networks. By the end of every $100,000$ cycles, each node would have executed its coloring algorithm an average of 8.09 times.

In accordance with the design goals laid out in Section 2.2, we monitor the number of defective edges present in the graph, the average number of connected components induced by each color, and the number of nodes which have been defined as being "vulnerable". The first metric is our primary optimization goal and corresponds to the number of edges that exist in the graph that can be traversed by a node-hopping attack. The second metric indicates the minimum number of separate infections that must take place for all vulnerable nodes to be compromised given an attack that is unable to change the color assignment. Since a separate curve exists for each color, we average the number of connected components across all colors for each algorithm analyzed. The final metric provides a baseline of the number of vulnerable nodes in the network. In the absence of an external agent, namely an attack that is aware of the coloring algorithm, this value should be affected only by the coloring algorithm itself.

Figure 4.13(a) shows the improvement in the number of defective edges as the three classes of dynamic algorithms converge to their local optimums. The difference in the quality of the solutions provided at convergence is shown in Figure 4.13(b). In Figure 4.13(c), a comparison of the number of average connected components for each color is presented. Figure 4.13(d) shows the evolution of the population of nodes of a single color; these nodes are later tagged as being vulnerable to attack and, if attacked, become malicious. The upward bias in the number of nodes of the specific color being examined is relatively small in comparison to the number of nodes on the graph and is an artifact of the simulation run. Not surprisingly, the number of nodes in the one color being examined is approximately the same for all three classes of algorithms.

In Figures 4.13(a) through (c), both the MUTUALLY BENEFICIAL SWAPPING and the GREATER GOOD SWAPPING algorithms provide an improvement as compared to the RANDOMIZED COLORING algorithm. The two swapping algorithms provide a solution which is inferior to the COLOR FLIPPING algorithm. The marked difference in the quality of the coloring solutions observed between the swap-based algorithms and the flip-based algorithm can be attributed to the availability of colors to any given node. In the swap algorithms, a node can only change its color to one that is present amongst its neighbors, and then only if the outcome of the swap is mutually beneficial to the nodes or globally beneficial to the graph. The flip algorithm places no restrictions upon a node's potential color choices if the node is exposed to a large number of monochromatic edges. As a result, the COLOR FLIPPING algorithm allows for a greater fraction of nodes to change their color assignment when the distributed algorithm is executed.

It is clear from Figure 4.13(b) that the RANDOMIZED HYBRID and BEST CHOICE

HYBRID algorithms produce a better coloring than either the swap-based or the flip-based algorithms alone. The hybrid algorithms generate a better solution by simultaneously drawing on the swap algorithm to eliminate deadlocks that may occur in a neighborhood and the flip algorithm to provide a wider range of colors that a node can assign itself.

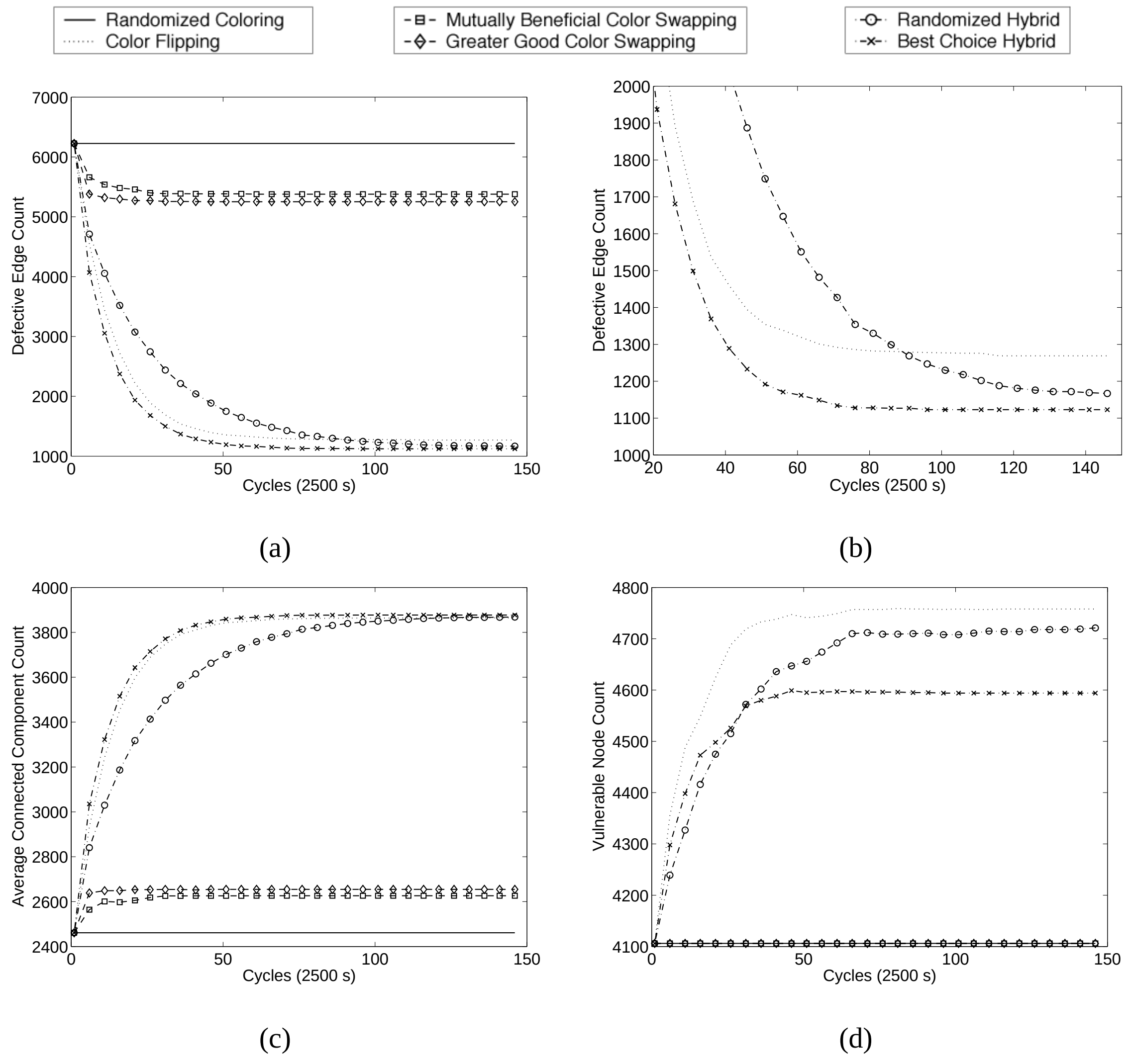


Figure 4.13: Comparison of coloring algorithms. The quality of the coloring, measured by both the number of defective edges and the number of disconnected components induced by the coloring, is maximized through the use of hybrid algorithms.

Chapter 5. Validating and Attacking Distributed Software Diversity

I think computer viruses should count as life. I think it says something about human nature that the only form of life we have created so far is purely destructive. We've created life in our own image.

Stephen Hawking

5.1 Validating Network Diversity Assignment Algorithms for Use in Virus Defense

The field of viral propagation modeling has garnered a great deal of attention in recent years as computer security researchers attempt to find ways of mitigating rapid malcode propagation. A variety of techniques have been suggested which can delay the spread of a worm, including rate-limiting network cards [89], targeted immunization of highly connected nodes [67], and a combination of address blacklisting and content filtering [61]. In complementary work, researchers have been focusing on the software monoculture on the Internet and its relationship to viral epidemics. The value of software diversity to computer security comes from the fact that an attack written for one piece of software rarely works for a different but functionally equivalent software package. By increasing the number of diverse software packages present on the network, the research argues, the chances that an attack will be effective against a randomly selected node will decrease.

The research literature in software diversity suggests that the introduction of different software packages is an effective method of disrupting the activities of an attacker or a worm, particularly one which repeatedly utilizes a pre-written and unchanging attack to compromise machines. However, there have been no quantitative evaluations of the impact of software diversity on malcode propagation in real network topologies. In this section, we use a popular metric called the *epidemic threshold* [87] to measure a network's resiliency against malcode propagation and study the steady state prevalence of computer viruses in the presence of software diversity. We show, through both modeling and simulation, that even a simple randomized distribution of diverse software packages can increase the epidemic threshold of both real and synthetically generated computer networks. This section also shows that an algorithm-driven distribution of diverse software packages, as discussed in Chapter 4 can further increase the epidemic threshold and serve as an effective method for preventing worm epidemics [64].

5.1.1 Problem Statement

In previous chapters, we showed that the location of diverse software packages on a network is as critical to effectively diversifying a network as the creation of diverse software packages. We have showed that the introduction of an alternative software package is able to reduce the number of edges across which a virus can traverse. Additionally, we showed that using an algorithm-driven diversity distribution is necessary for attempting to minimize the number of monochromatic edges, i.e., the number of homogeneous pairs of neighbors.

While the number of monochromatic edges is an effective metric as an optimization goal, it does not directly express the ability of the diversity assignment algorithm to limit the virulence of a worm. This section, on the other hand, quantifies the quality of a software diversity assignment by focusing on the effect that network assignments of diverse software has upon the propagation of worms. Given a worm whose rate of propagation from an infected node to each of its vulnerable neighbors is β and the rate at which infected nodes are disinfected is δ, we study the *epidemic threshold*, or the ratio of β/δ below which an infectious agent will burn itself out (i.e., the ratio below which there will be no infected nodes in the network at steady state).

One of the goals of any virus mitigation technique should be to increase the epidemic threshold of the network. In this chapter, our goal is to study:

1. The epidemic threshold with a randomized distribution of diverse software packages

to nodes in a real network (an IPv6 BGP topology) as well as a synthetically generated (an Erdös-Rényi random graph) network topology.

2. The relationship of the above results to the number of different software packages available to distribute among the nodes.

3. The epidemic threshold on the same networks with a topology-sensitive algorithm-driven distribution of diverse software packages.

As before, we represent a network of computers by graph G and a set of diverse software packages which can be assigned to nodes on the network by C. We consider a contagion which can infect only a single software package in C. Assume that the number of software packages available in C is greater than or equal to the chromatic number of the graph $\chi(G)$. If the software packages are randomly distributed to the network, then a portion but not all of the nodes will be rendered immune to the infection. However, if a graph coloring algorithm is used to assign the software in C to the nodes in G, then no edges will be left to spread the infection, and the infection is guaranteed to die out.

The remainder of this section is organized as follows. Section 5.1.2 describes related work in the fields of software diversity and viral propagation modeling. A generalized analysis of the viral propagation models and the impact of diversity upon them is presented in Section 5.1.3. In Sections 5.1.4 and 5.1.4.1, we extend both the statistically derived and graph theoretic viral propagation models to incorporate the impact of random as well as algorithm-driven diversity assignments. We validate these models using simulations of virus propagation on both synthetic and *real* network topologies. The simulations show that the improvement in the epidemic threshold experienced under an algorithm-driven diversity assignment algorithm is significantly higher than that predicted by the bounds generated by our models for real-world graphs.

5.1.2 Related Work

The research discussed in this chapter is based upon work from two independent but related fields, *software diversity* and *viral propagation modeling*. Software diversity research focuses on the creation and distribution diverse software packages to limit the exploitability of a security vulnerability by a worm, known as the *wormability* of a vulnerability [70]. While software diversity focuses on the interaction between a worm and a system at the moment of infection, the field of viral propagation research focuses on the modeling of large scale behavior of worms once they are established in the network.

5.1.2.1 Viral Propagation Modeling

All of the models discussed below are based upon the SIS, or Susceptible-Infected-Susceptible paradigm, where the individual vertices on a graph are either in one of two states: *susceptible* to infection or *infected*. A node moves from the susceptible state to the infected state when an infected neighbor, with the probability β, passes on its contagion. A node moves from the infected state to the susceptible state, independent of its number of neighbors, with the disinfection probability δ. As discussed before, if the ratio β/δ is below the epidemic threshold, the infection will eventually die out.

Kephart and White [49] considered viral propagation on a Erdös-Rényi random graph in an early contribution to the study of computer virus epidemiology. Their assumptions regarding the homogeneity of the nodes in the communication network allowed the authors to model the behavior of an infectious agent using a first order differential equation. A steady state solution for the differential equation is found which provides a bound on the epidemic threshold as a function of the average degree in the graph. They show that the ratio of the infection rate to the disinfection rate must be less than the inverse of the average node degree, $\langle k \rangle$, in order to prevent an epidemic:

$$\frac{\beta}{\delta} < \frac{1}{\langle k \rangle}$$

Pastor-Satorras and Vespignani produced a model which provides insights into the propagation of viruses on graphs with arbitrary degree distributions [65]. Their analysis provides a bound on the epidemic threshold in terms of the node degree's first and second order statistics:

$$\frac{\beta}{\delta} < \frac{\langle k \rangle}{\langle k^2 \rangle}$$

The authors leveraged statistical mechanics to determine closed-form expressions of the second-order statistics of degree distributions for specific classes of graphs. When applied to synthetic graphs which are statistically similar to real world networks, the model predicts that every infection will become an epidemic as the number of nodes tends to infinity. On graphs sampled from real-world data, the number of nodes is finite, and while small, the epidemic threshold is non-zero and can be evaluated numerically.

Y. Wang and others [86] created a discrete time model which is then converted to vector-space notation, which encapsulates the infection state of each node on the network. Using algebraic manipulation, they isolate a system matrix which determines the current infection state based upon the previous system state using a method similar to solving discrete time Markov chains. Spectral decomposition bounds the epidemic threshold of a virus propagating on the network to the inverse of the largest eigenvalue of the graph's adjacency matrix $\mathbf{A}$:

$$\frac{\beta}{\delta} < \frac{1}{\max\limits_{\forall i}\{\mu_i(\mathbf{A})\}}$$

Each of the different models examined here have seemingly different methods to determine the upper bound on the epidemic threshold. For the Kephart and White model, the epidemic threshold can be maximized by minimizing the average number of adjacent systems which are vulnerable to a worm. The epidemic threshold will be increased in the Pastor-Satorras and Vespignani model by minimizing the second order statistic while maximizing the first order statistic. The Wang model's epidemic threshold can be maximized by minimizing the largest eigenvalue of the diversified network's adjacency matrix. We show, however, through modeling and simulation, that all of these goals can be met by reducing the number of edges across which a virus can traverse in a diversified network.

5.1.3 Viral Propagation and Software Diversity

It is possible to show that, regardless of the underlying viral propagation model, an assignment of software packages to a graph such that the assignment forms a perfect graph coloring will force the epidemic threshold to infinity. Consider a perfect coloring, where there are no edges across which a virus can propagate. The only infected hosts that exist are those which are initially infected by a virus. Because this set cannot increase, the disinfection rate of systems will continually decrease the number of infected systems until all systems are uninfected.

As pointed out in Chapter 4, it may not be possible to guarantee that a sufficient number of software systems will be available to perfectly color the network. It would then be more appropriate to assign the limited amount of diversity so as to limit the number of monochromatic edges and thus increase the epidemic threshold. To achieve this goal, we use the COLOR FLIPPING algorithm described in Section 4.2.2. The distributed algorithm has each node choose an initial software package, or color, and, at random intervals, communicate with their immediate neighborhood of nodes to discover their current color. The node initiating the communication will then switch to the neighborhood's minority software package if it finds a majority of its neighbors are running the same software package.

It is important to note that it is not necessary to examine every variation of the coloring algorithms discussed in Chapter 4 for their effect on the viral propagation metrics. We are interested in showing the trends that a decreasing defective edge count has upon the studied viral propagation characteristics, which will be provided by any of the distributed algorithms.

The introduction of a graph coloring algorithm removes some of the assumptions of randomness that underpin the statistical models discussed, which results in loose bounds on the epidemic threshold on networks colored using the COLOR FLIPPING algorithm. Rather than providing only loose bounds, we examine the effect of algorithm-driven color assignments on the epidemic threshold primarily through the use of simulation.

5.1.4 Statistical Models

We can consider nodes which run software packages which are different from their neighbor to be relatively immune to attack from their neighbor. Assuming a randomized distribution of diverse software packages, if there are c software packages available for n nodes, it is expected that $n - n/c$ nodes will be relatively immune to the n/c vulnerable nodes.

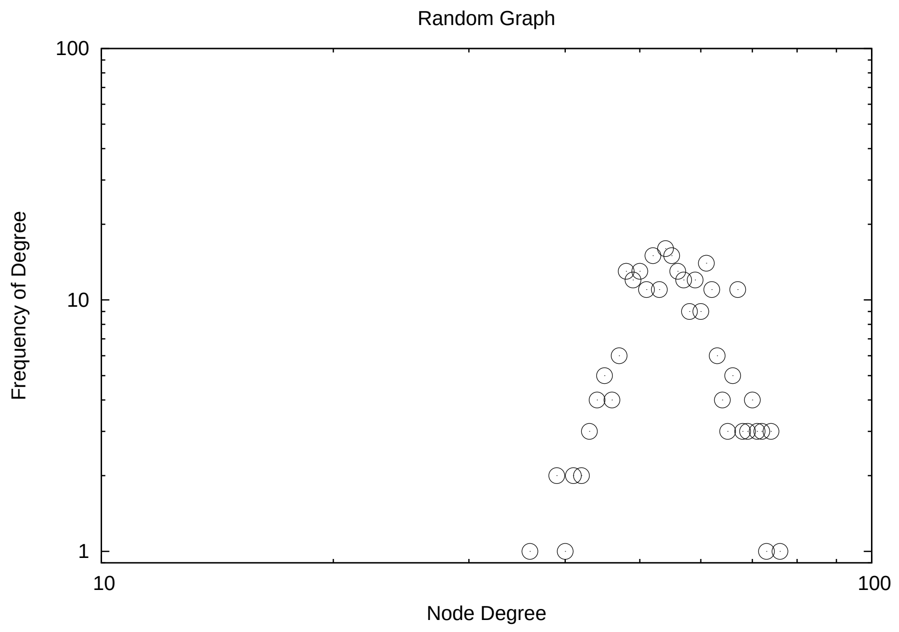
The effective infection rate, or the rate any given infected node can infect a neighboring homogeneous node, becomes:

$$\beta' = \beta \frac{1}{c}$$

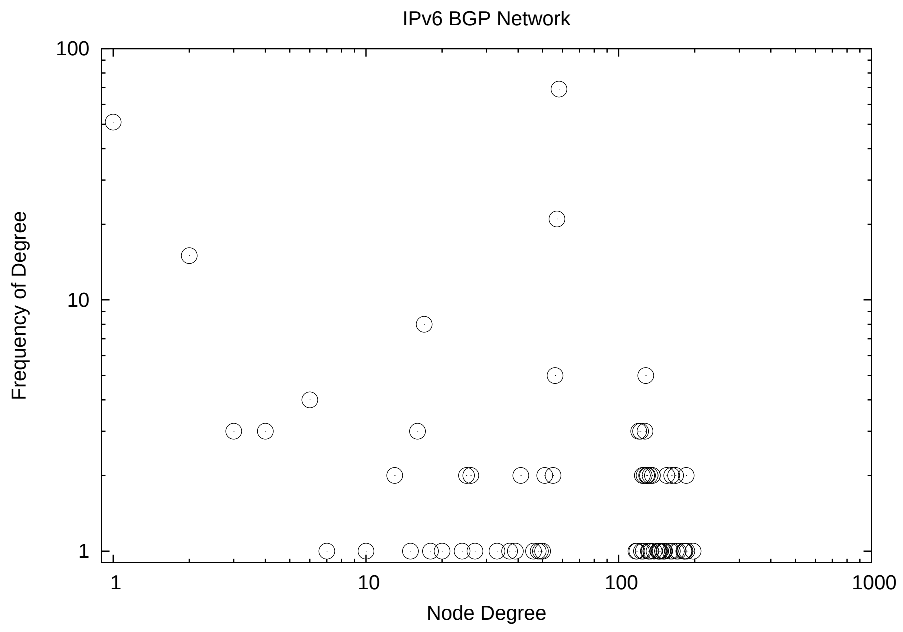
with an epidemic threshold given by:

$$\frac{\beta}{\delta} < \frac{c}{\langle k \rangle}$$

For a given β and δ, the critical number of software packages needed to ensure that a worm infection does not become an epidemic is:



(a)



(b)

Figure 5.1: Plot of the degrees of nodes found in the examined networks versus the frequency of the occurrence of the degree. The graph examined in (a) was constructed from a standard random graph model, and contains 266 nodes and $7,448$ edges. The graph examined in (b) was sampled from the IPv6 BGP topology, and contains a similar number of nodes and edges.

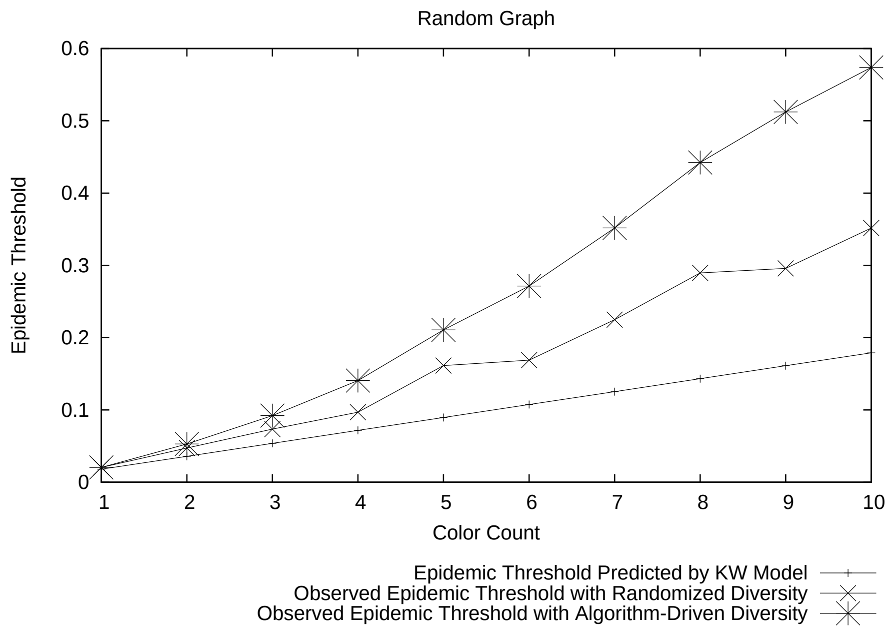
$$c_{crit} = \left\lceil \frac{\beta}{\delta} \langle k \rangle \right\rceil$$

A similar analysis can be done for the Pastor-Satorras and Vespignani model, which shows an increase in the epidemic threshold by a similar factor.

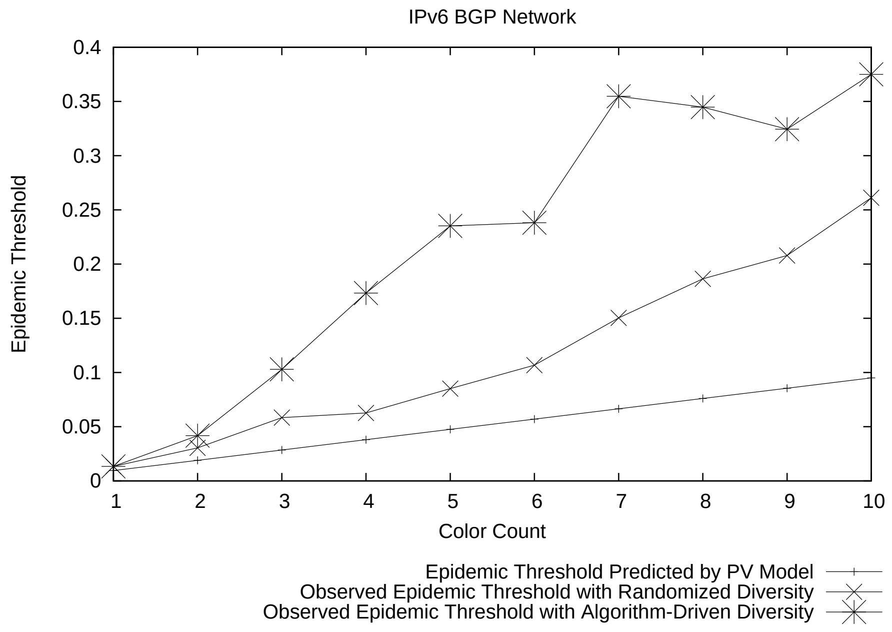
In order to test the utility of diversity assignments for increasing the epidemic threshold, it is necessary to either generate or measure a network topology for simulation study. Our first network was generated by collecting a list of the BGP peers present in the IPv6 network by accessing the routing table from IPv6 capable Looking Glass routers. A second network was created using an Erdös-Rényi random graph generator. Both graphs contain 266 nodes and approximately $7,500$ edges. The distribution of the individual node degrees is shown on a log-log scale in Figure 5.1. While both graphs have similar average degree, the degree distribution for both graphs is dramatically different. The distribution plot of the synthetic graph, shown in Figure 5.1(a) corresponds to a standard random graph, while the distribution of the sampled graph's topology, shown in Figure 5.1(b), shows the same self-similar characteristics that have been observed in previous literature [33]. While we utilized larger networks to study the coloring algorithms in Section 4.1.2, the computational load associated with executing the viral propagation simulations makes this option infeasible.

The rest of the simulation studies presented in the section follow a standard methodology; a single color is tagged as being vulnerable to infection, and the graph is assigned an initial coloring. A high percentage of the nodes assigned the vulnerable color are randomly chosen to be the nodes which initially contain the infection. We experimentally determine the epidemic threshold by progressively changing β relative to a fixed δ until a persistent infection is not seen over numerous simulation runs with both the same initial infection set and with alternate initial infection sets.

The simulation exercises shown in Figures 5.2(a) and (b) examine the effect that the



(a)



(b)

Figure 5.2: Comparison of the effect of the number of colors on the experimentally determined epidemic threshold. In both (a) and (b), a graph is assigned either one color for every node, multiple colors via a randomized algorithm, or multiple colors via the described COLOR FLIPPING algorithm. It can be seen in both graphs that the epidemic threshold increases as the diversity-assignment algorithms become progressively more sophisticated.

number of colors has upon theoretically derived and experimentally-evaluated epidemic thresholds. For each color, we compute the diversity-aware variants of the Kephart and White (KW) model and the Pastor-Satorras and Vespignani (PV) model presented in Section 5.1.4 for the random graph and the IPv6 graph, respectively. Additionally, both randomized and algorithm-driven color assignments, based upon the COLOR FLIPPING algorithm presented in Chapter 4, are performed on the graph for each color count examined.

The data shows that the bound on the epidemic threshold of a randomized coloring provided by the statistical models is below the experimentally determined epidemic threshold. The result allows us to conclude that the epidemic threshold of a diversified network will be higher than the epidemic threshold of a homogeneous network even if diversity is assigned randomly. It is noteworthy that the epidemic threshold is significantly increased by allowing an algorithm to assign diverse software packages to nodes on the network; this leads us to conclude that a planned diversity assignment is a worthwhile undertaking in order to maximize the epidemic threshold of a network.

5.1.4.1 Graph Theory Derived Models

In a fashion consistent with Wang's model, we are able to restate the goal of the software assignment in terms of graph partitions and the subsequent eigenvalues of the subgraphs. We denote our software assignment as $f : V(G) \mapsto C, C = \{1, 2, ..., c\}$, where C is the set of available software packages. Let $G_i := G[f^{-1}(i)] : i \in C$, where G_i are the subgraphs induced by color i. Define $\mu_{max}(G_i)$ as the maximum eigenvalue of subgraph G_i's adjacency matrix. Therefore, we wish to find a software assignment f_{opt} where:

$$f_{opt} = \arg\min_{\forall f} \left\{ \max_{i \in C} \{\mu_{max}(G_i)\} \right\}$$

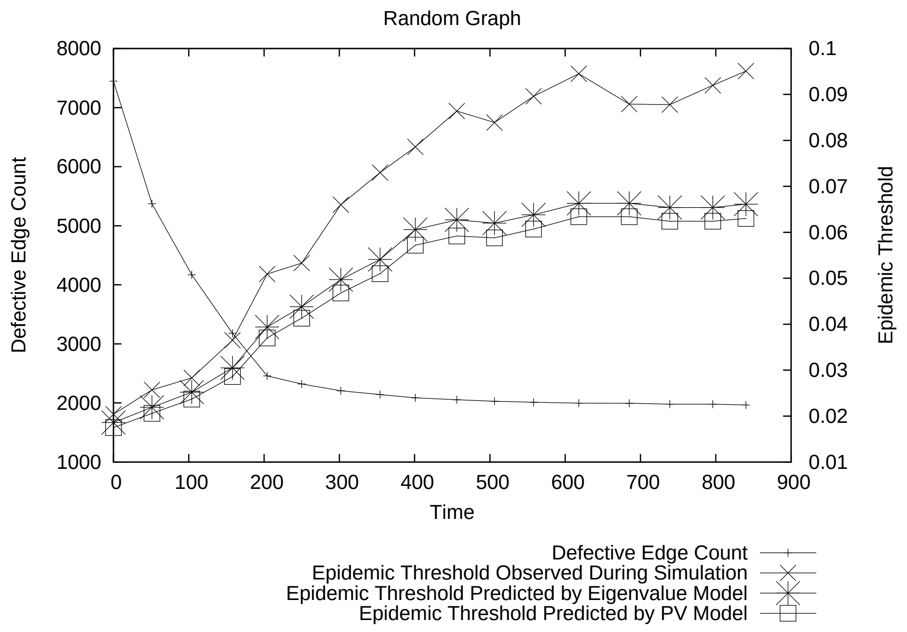
which minimizes the maximum eigenvalue across all subgraphs induced by each color.

Loose bounds for general graphs and hard bounds on regular graphs can be determined for the largest eigenvalue of the adjacency matrix of a diversified network. Rather than relying upon the loose bounds, we directly measure the eigenvalue of a network which is actively undergoing diversification to predict the epidemic threshold.

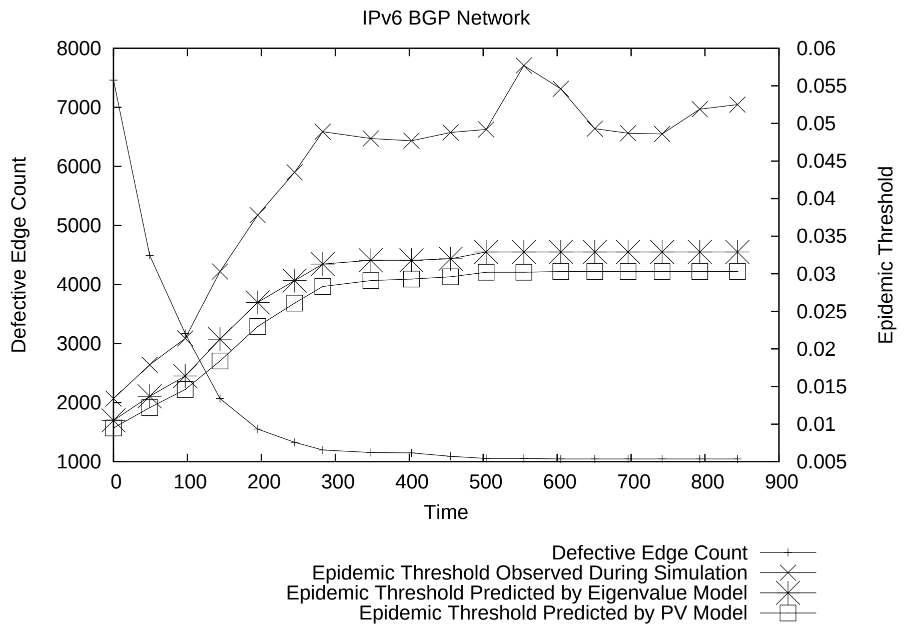
To examine the impact the number of monochromatic edges has upon the epidemic threshold, we simulate a homogeneous network of systems, then allow each system to minimize its number of monochromatic neighbors by executing the COLOR FLIPPING algorithm presented in Section 4.2.2. At each time-step, we compute the epidemic threshold predicted by the Pastor-Satorras and Vespignani model and Wang's eigenvalue model. The Kephart and White model is inappropriate for use with networks using an algorithm-driven diversity assignment as the application of the algorithm to the network removes the homogeneous degree distribution on the network.

Figures 5.3(a) and (b) show the impact that decreasing the number of monochromatic edges has upon the statistical, eigenvalue-derived, and experimentally found epidemic thresholds. It is clear from the simulation studies that reducing the number of monochromatic edges in the network is an extremely effective method of increasing the epidemic threshold. The simulation studies confirm the utility of recomputing the eigenvalue-derived epidemic threshold with each step of the graph coloring operation is an effective method of approximating the epidemic threshold. Furthermore, the experiment shows that decreases in the number of defective edges go hand in hand with increases in the epidemic threshold.

While a wide variety of techniques for mitigating rapid malware propagation have been analyzed and simulated using standard virus modeling techniques, the contributions of the software diversity community have not yet been fit into this framework. In this section, we make the first contributions toward analyzing viral propagation modeling in the presence of software diversity. We use both models and simulations to show that on both simulated and real networks of systems, a naïve, randomized software diversity assignment is able to increase the epidemic threshold. Simulations also show that an algorithm-driven diversity assignment is able to further increase the epidemic threshold beyond that seen with a ran-



(a)



(b)

Figure 5.3: Comparison of the effect of the number of defective edges on the epidemic threshold. In both (a) and (b), the nodes of the graphs all begin at the same color, and the COLOR FLIPPING algorithm is executed to find a 3-color assignment which reduces the number of monochromatic edges. As the number of monochromatic edges decreases, the experimentally determined epidemic threshold increases beyond what is predicted by statistical models and by the eigenvalue model.

domized assignment. These results provide quantitative insight into the impact of software diversity on the tolerance of a network to viral attack.

5.2 Attacking Network Diversity Assignments

In the previous section, we confirmed through simulation and analysis that reducing the number of defective edges directly increases the epidemic threshold of a network. It is likely that an virus would be interested in affecting the performance of the coloring algorithm itself, given that the distributed algorithms discussed are being used to decrease the ability of an attacker from compromising the network.

We propose a set of primitive behaviors exhibited by a malicious node from which any attack can be created:

Spreading Upon inspection, instead of looking to flip its color, a node that is malicious will look to subvert a neighboring node that is of its own color.

Misrepresentation A node may falsely report its current color when it is queried for its color by neighboring nodes. Additionally, a node may falsely report its defective edge reduction to neighboring node wishing to conduct a color swap.

Inertia A node will not change its color regardless of external stimulus.

Each of these attacks are presented in pseudocode format in Figure 5.4. These functions are written in such a way that they can be directly incorporated into the algorithms presented in Chapter 4.

The first algorithm analyzed is robust against attacks directed toward the algorithm itself. The RANDOMIZED COLORING algorithm requires nodes to set their color without examining their environment. In turn, any network implementing the algorithm is not affected by the last two attacks, and can only be affected by the spreading attack.

```
# Spreading Attack:
**EventLoop()**:
    **if** the timer event has occured:
        Attack any neighboring node of the same color
        Set new timer event
    Continue **EventLoop()**

# Color Liar Attack
**RespondToColorQuery(**Neighbor**)**:
    Transmit a randomly chosen color from $U - CurrentColor$ to Neighbor

# Defect Liar Attack
**RespondToSwapImprovementQuery(**Neighbor**)**:
    MyDefect := **ComputeDefect()**
    NewDefect := **ComputeSwappedDefect(**Neighbor**)**
    DefectImprovement := MyDefect - NewDefect
    Transmit a large random value to Neighbor

# Swap-contract Breaking Attacks
**RespondToSwapRequest(**Neighbor, NeighborColor)**)**:
    Transmit AcceptedRequest to Neighbor
    Do not change the current software package
```

Figure 5.4: Pseudocode used by an attacker wishing to compromise a network of hosts running the distributed coloring algorithms presented in Chapter 4.

The COLOR FLIPPING algorithm introduces an inherent security flaw. Any node looking to flip its color must trust that their neighbors will be truthful in reporting their own color assignment. If a malicious node decides to lie about its own color, it can influence a querying node's color choice, but not force a color assignment upon the querying node. For example, a malicious node can falsely report to a node that its color is the same as a querying node, which would contribute to the querying node's defect count. If the malicious node is fortunate, the defective edge count observed by the querying node would become greater than $\lfloor d(v)/k \rfloor$. This will cause the querying node to flip to a new color. The goal of the malicious node is to push the querying node to flip to a specific vulnerable color. If a flip takes place, the malicious node has no way of being certain the querying node will flip to a vulnerable color.

Both the MUTUALLY BENEFICIAL SWAPPING and GREATER GOOD SWAPPING algorithms introduce a security flaw due to the inherent trust associated with a color swap. If a malicious node either proposes or agrees to a swap with a participating neighbor, it can keep its own color even after the neighbor has completed switching to the new color. The action would create a defective edge that the malicious node can use to propagate an attack. In the case of the mutually beneficial swap algorithm, a swap would never be acceptable to a node unless the defective edge count of the node decreases. Even if a malicious node wants to "push" a vulnerable color onto a node, it would only be able to do this to the subset of its neighbors which would stand to gain from an honest swap. The GREATER GOOD SWAPPING algorithm, however, has a larger security vulnerability associated with it. A malicious node can force a color change onto a neighboring node by claiming an extremely high defect improvement. To the neighbor, it would appear that the proposed swap is globally beneficial, regardless of its own increase in the number of defective edges. Therefore, a single compromised node can spread a chosen color across an entire network, one node at a time.

There does not exist a single optimal attack that works against both algorithms, however. If the network implements a swapping algorithm, lying about a malicious node's own color would lead a querying node to swap to a random, non-vulnerable color. Rather than increasing the number of nodes that can be attacked in the network, running the optimal swapping algorithm attack on a network running the color flipping algorithm would actually *decrease* the number of vulnerable nodes. Vulnerable nodes, which were previously unable to swap their color to one which would induce less defective edges because of a lack of potential swapping partners would find nodes with a previously unseen color in their neighborhood. Therefore, not only would the number of vulnerable nodes decrease, the number of defective edges present across the network would decrease as well. Likewise, a network running the color flipping algorithm would not be impacted by the contract-breaking attack mentioned above. No inter-node contracts are involved in the algorithm, and correspondingly, there is no opportunity to break a color-changing agreement.

Based upon this analysis, the behavior of the hybrid algorithms discussed in Section 4.2.4 under attack can be expected to be a synthesis of the reactions of both the color swapping and color flipping algorithms to the stated attacks.

5.2.1 Attack Simulation

A second series of experiments is conducted to test each algorithm's tolerance to attack. One color is selected and labeled as *vulnerable*, meaning an attacker can compromise that color and only that color. It then becomes the goal of the attacker to switch every node in the network to the vulnerable color. After the coloring algorithms have converged, 1% of the vulnerable nodes are infected with a worm, which is able to carry out any combination of the attacks described in Section 5.2.

It is not necessary to recompute the epidemic threshold at each point of the virus' progression throughout the network. As we have shown in Section 5.1.4.1, the defective edge count is a sufficient metric for measuring the epidemic threshold of a graph.

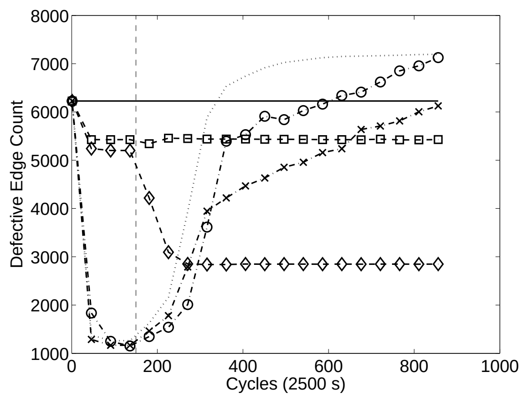
Figures 5.5(a), 5.5(b), and 5.5(c) show the effect of malicious nodes on the number of defective edges present, the average number of connected components for each color, and the number of vulnerable nodes, respectively. These malicious nodes are introduced to the network after the distributed algorithm has largely converged. They begin to attack the network by lying about their color and breaking swapping contracts, but respond honestly when asked about their own improvement with respect to the number of similarly colored neighbors when queried about a proposed color swap.

Figures 5.6(a), 5.6(b), and 5.6(c) show the effect on the metrics studied in Figures 4.13 and 5.5(a)–(c) when nodes that lie about the quality of a proposed swap and break swapping contracts are introduced into the network some time after convergence. It should be noted that the COLOR FLIPPING algorithm is not vulnerable to this attack, since it does not propose swaps with neighboring nodes.

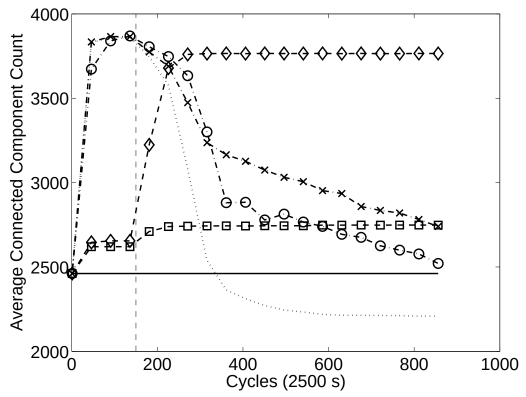
Figures 5.7(a), 5.7(b), and 5.7(c) show the effect of completely dishonest nodes upon the network. This "brute force" attack is not designed to attack any one particular algorithm, nor are the malicious nodes cognizant of the coloring algorithm that is being executed by their neighbors. Instead, it is designed to examine the effects of completely uncooperative nodes upon the network.

As stated in Section 5.2, color liars increase the number of defective edges in a network when the network is executing the COLOR FLIPPING algorithm, but decrease the number of defective edges present in a network executing the COLOR SWAPPING algorithms. The introduction of color liars in Figure 5.5(a)–(c) experimentally confirms this analysis. The behavior of the hybrid algorithms indicates a bias in both algorithms towards the use of the COLOR FLIPPING strategy as opposed to the COLOR SWAPPING strategy, as evidenced by the similarity between the number of defective edges experienced by the COLOR FLIPPING, RANDOMIZED HYBRID, and BEST CHOICE HYBRID algorithms in 5.5(a). Furthermore, the experiment has shown that even after convergence is achieved, it is possible to disrupt the color assignment of the graph.

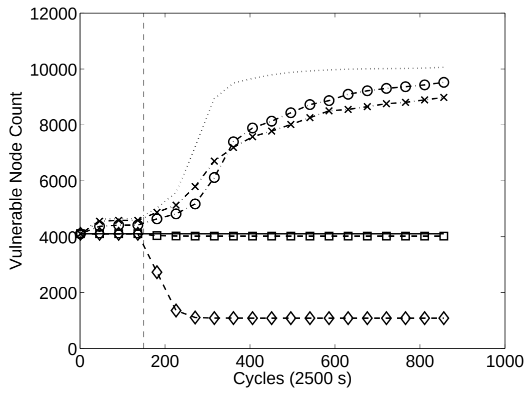
The behavior of a network that is being attacked via defect liars is dramatically different, as shown in Figure 5.6(a)–(c). While the network implementing MUTUALLY BENEFICIAL SWAPPING algorithm appears to not be affected by the malicious behavior, the network utilizing the GREATER GOOD SWAPPING is completely compromised. The two algorithms, while exceedingly similar, exhibit markedly different tolerance to attack. The rationale for this phenomenon resides in the relative "voting power" of swapping partners.



(a)



(b)



(c)

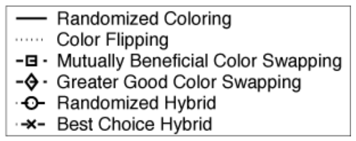
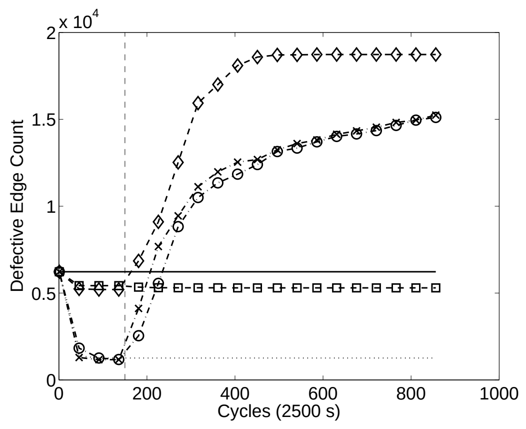
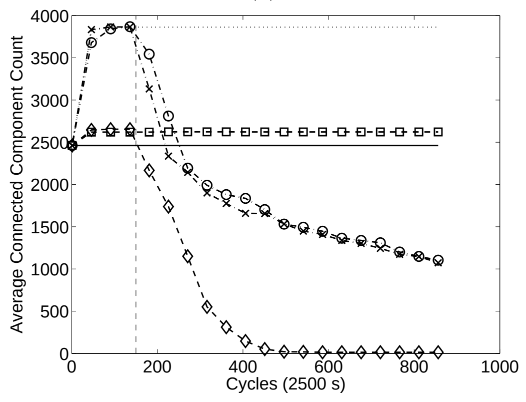


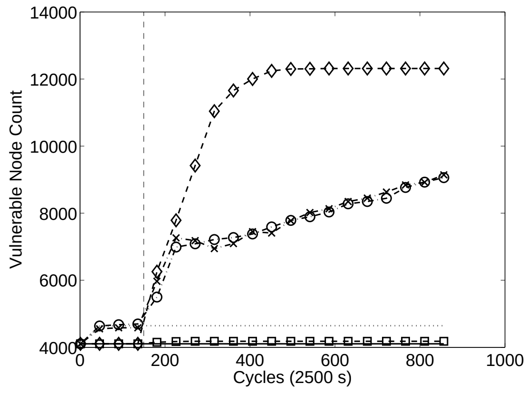
Figure 5.5: Comparison of the impact of nodes that only lie about their color on the distributed algorithms. The vertical line indicates the time when malicious nodes are added to the network.



(a)



(b)



(c)

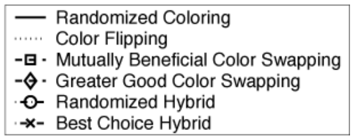
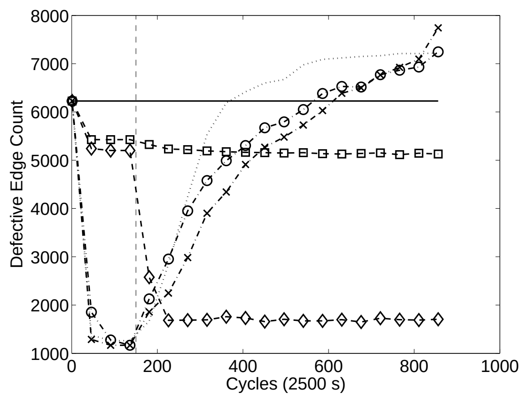
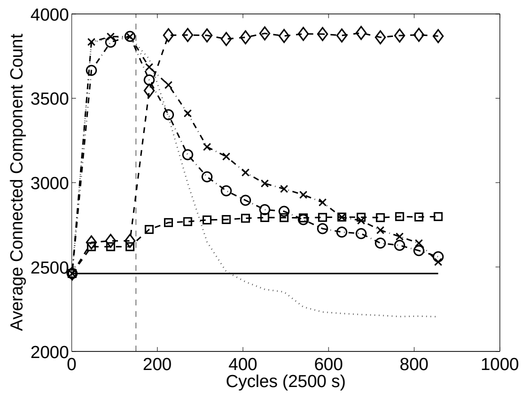


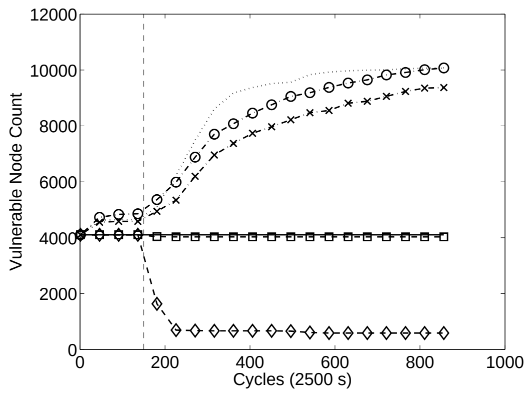
Figure 5.6: Comparison of the impact of nodes that only lie about their defect improvements on the distributed algorithms. The vertical line indicates the time when malicious nodes are added to the network.



(a)



(b)



(c)

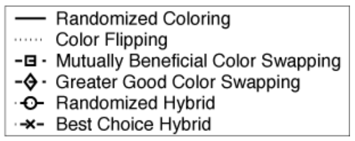


Figure 5.7: Comparison of the impact of nodes that only lie about both defect improvements and their color on the distributed algorithms. The vertical line indicates the time when malicious nodes are added to the network.

In the MUTUALLY BENEFICIAL SWAPPING algorithm, both neighbors have equal input for the swap decision. Regardless of the input of one's neighbor, a swap will not take place unless the action can benefit both nodes. After the distributed coloring converges, no node operating under this algorithm can further improve the quality of its coloring by conducting a swap. Nodes in networks implementing the GREATER GOOD SWAPPING algorithm, however can *always* conduct a swap that the node *believes* would increase the quality of the network's coloring. Under this algorithm, a swap partner can have an unbounded contribution to the swap decision. A malicious node can use this to force a coloring upon any neighboring node with whom a swap is being negotiated. Since the hybrid algorithms depend upon the swapping algorithm, they are both vulnerable to this form of attack.

In the plots contained in Figures 5.5, 5.6, and 5.7, the performance of the RANDOMIZED HYBRID and the BEST CHOICE HYBRID algorithms under attack appear to be rather similar. As stated in Section 4.2.4, the RANDOMIZED HYBRID algorithm contains a tunable parameter, however, which forces the algorithm to utilize the COLOR FLIPPING algorithm at a higher or lower frequency compared to the GREATER GOOD SWAPPING algorithm. Deriving the optimal balance between the two algorithms for the purpose of minimizing the effects of an attack against the algorithm can be accomplished using game theory, but the equilibrium point would be unique to the topology of the graph. In the formulation, payoffs experienced by either the network operator or the attacker would be derived from the rate at which non-vulnerable nodes can be convinced to change to a vulnerable color because of input from malicious neighbors. The usage rate of either the COLOR FLIPPING or the GREATER GOOD SWAPPING algorithms would be selected to balance out the risk of executing either of the two algorithms over the long term.

5.2.2 Analysis of Simulation Results

Two important conclusions can be drawn from the analysis of the coloring algorithms and their tolerance to a tailored attack. The MUTUALLY BENEFICIAL SWAPPING algorithm converges to the largest number of defective edges of any algorithm which allows for re-coloring of individual nodes. After convergence, though, attacking this algorithm has shown to be extremely difficult. A slight modification to the MUTUALLY BENEFICIAL SWAPPING algorithm was presented in the GREATER GOOD SWAPPING algorithm, which relaxes the guidelines for an acceptable swap. While this allows for more color swaps to take place and in turn reduces the number of defective edges in the graph, the algorithm becomes far more vulnerable to a directed attack. Algorithms which allow a node to undergo a local and independent color flip, while extremely effective at reducing the total number of defective edges, have been shown to be heavily impacted by malicious nodes which lie about their color. Given enough time for convergence and a small but finite set of moderately connected nodes, the malicious nodes would likely be able to compromise the entire network. The only algorithm which is not vulnerable to a directed attack is the randomized algorithm, which, not coincidentally, provides the worst defective coloring performance. Based upon these results, we believe that *there is a fundamental tradeoff between the quality of the diversity achieved by an algorithm and the algorithm's tolerance to attacks.*

Both of the hybrid algorithms allow for a node to choose between the two coloring algorithms at each instant of operation. The ability to switch between the two algorithms removes the attacker's ability to know which coloring algorithm a targeted node is intending to execute. In the absence of precise knowledge of the currently running coloring algorithm, an attacker would have some difficulty crafting an optimal attack. As discussed in Section 5.2, the most effective attack against the COLOR FLIPPING algorithm would be the introduction of color liars to the network, and the most effective attack against the swapping components of both the MUTUALLY BENEFICIAL SWAPPING and the GREATER

GOOD SWAPPING is the introduction of contract breakers to the network. For a contract-breaking node to work correctly, however, it must be completely honest about its color. Otherwise, the swap partner would swap to a different color from that of the malicious node, which would halt the proceeding attack. Through similar reasoning it is easy to see why introducing a set of honest contract breakers would be counterproductive for attacking all coloring algorithms.

The above rationale is no different from the motivation for security through diversity itself. The COLOR FLIPPING, MUTUALLY BENEFICIAL SWAPPING and GREATER GOOD SWAPPING algorithms are vulnerable to attack simply because the same algorithm is running on every node, and every node is vulnerable to the same form of attack. Introducing diversity at the diversity assignment layer would mean an attacker would not be able to use a single attack strategy to take over the network. The RANDOMIZED HYBRID and the BEST CHOICE HYBRID algorithms are vulnerable to all forms of misrepresentation and contract-breaking attacks, but the existence of a mixed coloring strategy increases the algorithm's tolerance to attack. Experimental evidence has shown that both hybrid algorithms fare better when presented with both forms of attack, than the COLOR FLIPPING and GREATER GOOD SWAPPING algorithms when each are presented with their appropriate attack strategies. The increased tolerance to attack is due to the lack of knowledge on the part of the malicious nodes; since the malicious nodes are unaware of which algorithm is being executed by the targeted nodes, choosing an effective attack becomes a game of chance. It is based upon this observation that we state that *the most effective way of achieving attack tolerance in our algorithms is to reapply the fundamental motivation of this dissertation, and implement diversity strategies into the algorithms themselves.*

Chapter 6. Conclusion and Future Work

Factum est illud, fieri infectum non potest.

Plautus

6.1 A Summary

Despite the best efforts of the computer security community, software with hidden vulnerabilities is still released into the world. Improved lexical analyzers and hardened operating environments have helped to reduce the exploitability of software. Better education and experience has created attentive administrators, who in turn have cut down on the risk of a software vulnerability being used as a worm vector through attentive patching and network access control. Even with the number of easily exploitable vulnerabilities decreasing and the window of a disclosure's value slowly being closed, the two research paths will never eliminate the possibility of intrusion. It is in the gap between pre-vulnerability prevention and post-vulnerability mitigation techniques that researchers have proposed presenting attackers with a diversity of systems as a target as opposed to something familiar.

The application of this *software diversity* paradigm requires the acquisition of diverse software packages which are relatively immune to common faults. The generation and management of this diversity is an active research area. The literature details a variety of solutions, including introducing heterogeneous software to systems through randomization of memory structures, N-version programming, and various other techniques. However, for both business and technical reasons, the limited number of functionally equivalent yet distinct software packages makes diversity a less effective strategy than one may like.

In this dissertation, we have provided a general model for software diversity by considering how techniques for generating diversity combine and present themselves to attackers.

The model shows that an algorithm for diversity assignment is required in networks of systems where the heterogeneous software set is small in order to maximize the inherent value of diversity. The analysis inspired us to provide a series of algorithms for increasing the effectiveness of system-level heterogeneity on a network. Even though the computation of an optimally diverse software allocation is believed to be intractable, the distributed algorithms presented here reduce the number of links that can be utilized for propagating an attack. Furthermore, our algorithms effectively cluster the network, which helps to isolate infected systems from the rest of the topology.

Decreasing the number of edges an attacker can traverse is relatively abstract concept that does not appear to directly translate into an quantifiable impact on an attacker's performance. We have successfully been able to link this defective edge count to the positive impact of our diversity assignment strategies against attackers using both analysis and simulation techniques borrowed from the field of viral propagation modeling.

Any methodology for increasing the attack tolerance of a network is destined to come under attack itself. We have shown that there exists a trade-off between the ability of an algorithm to reduce the number of defective edges present in the network and the ability of the algorithm to tolerate a directed attack. The algorithm which exhibits the best worst-case performance against attack was a hybrid of our basic algorithms, which itself highlights the principle of security through diversity.

Based upon our observations, simulations, and analysis we are left with a confirmation of our thesis statement; not only is diversity critical for improving the attack tolerance of a network, but the inherent value of diversity can be increased through an algorithmic distribution of diverse systems. Furthermore, these principles must be applied to all levels of system design, including any scheme which introduces diversity itself.

6.2 Future Work

Multiple problems have been identified during the course of the research related to network diversity assignments whose solutions fall outside the scope of this dissertation. In the following sections, we outline the measurement experiments which must be performed in order to correctly quantify the cost and effectiveness metrics associated with each diversity and attack technique. We additionally describe the engineering work that one can undertake to implement a network of diverse systems using largely off-the-shelf technologies.

6.2.1 Measurements and Data Analysis

As stated in the concluding remarks, the metrics and functions associated with the diversification hypergraph presented in Chapter 3 are difficult to analytically quantify. A set of measurement experiments should be conducted in order to provide estimates for a number of these metrics.

Consider the following initial experiment. Information on the global background security events that are associated with worm traffic can be gathered using network telescopes [60], or routable, unpopulated, and contiguous address spaces which are monitored by traffic monitoring software. A tool which examines packet-level idiosyncrasies to determine the traffic source's operating system can be used to determine the characteristic of a source of an attack without directly contacting the system. These pieces of software, known as a passive OS fingerprinters, are freely available for download today [81]. This information can be coupled with attack fingerprints culled by a signature-based IDS to determine source and target operating system pairings. The vast number of both Microsoft-targeted worms and Microsoft-based attack sources will skew the data collection towards a single platform, in turn necessitating a lengthy data collection period. We predict that the end result of this basic exercise will show an extremely high correlation between the software running on the source of the attack and the software running on the target of the attack, which would be a validation of the assumptions made in Section 4.1.2.

The previously mentioned experimental setup does not differentiate between an attack attempt made by an intelligent attacker or by an automated worm. It is likely that an intelligent attacker would completely ignore a network telescope-based measurement scheme due to the lack of responses elicited from any input traffic. A minimal response to input traffic can be generated using software which presents itself as a host with several open ports, but does not respond with anything beyond basic information banners. Niels Provos' honeyd package currently provides this functionality [71]. Separating out worm traffic would be challenging, but could possibly be done through statistical means by comparing attacks against two honeyd systems, where one system provided banners consistent with the host fingerprint and the other system did not. Intelligent adversaries would attempt to attack the application specified by the modified banner, while worms would most likely continue a standard attack. This measurement experiment is still an approximation of the behavior of the adversary. It is impossible to know the adversary's true nature and capabilities without allowing the attack to be fully carried out.

We can safely allow an adversary to execute an attack against a machine under our control using heavily instrumented systems placed in specially administered and monitored sections of a network. These *honeypots* have been used heavily in the past for determining the relative skill and sociological behaviors of computer hackers [82]. Our experiments can use an array of honeypots to generate statistically significant measurements on the rate of infection by worms from a similar source host as compared to intelligent attackers operating from similar sources.

Using a large sample of honeypots would be a viable method of testing the effectiveness of each practical diversity technique in the literature. The sample machine's subnet can be cut in half with half the machines being diversified and the remaining half being stock systems. The mean-time-to-successful-attack figures can then be used to build the effectiveness function discussed in Chapter 3. Similar experiments can be conducted to determine the effectiveness of combinations of multiple diversity techniques.

The amount of time required to conduct these experiments is somewhat unbounded. A significant number of determined hackers have to approach and compromise the systems before meaningful statistics can be collected. Nevertheless, such experiments would be useful for determining not only the true value of software diversity but also the metrics needed for optimal construction of both host-level and network-level diversity assignments.

6.2.2 Implementation Strategies

The diversity assignment algorithms presented in this dissertation are not purely theoretical constructs. They can be implemented using currently available, off-the-shelf hardware and software systems. A prototype network of diverse systems can be built out of standard commodity hardware running kernel hypervisors and virtual machines similar to Xen [7] and VirtualPC. The hypervisor can be loaded with several different operating systems, and each system can then be set up using a standardized configuration assignment tool such as CFEngine [72]. A diversity of assignment algorithms can then be implemented at the hypervisor level, where the challenge of rapidly switching a host's "color" can be solved by allowing the hypervisor to activate and deactivate each of its child virtual systems depending upon the demands of the assignment algorithm.

As discussed in Section 2.2.1, sensor network technology is a perfect candidate for the presented diversity assignment schemes. Sensor network nodes are comprised of limited-capacity processors, transceivers, and sensors. A network designer can heavily specify the interconnection protocol and sensor packages carried by each node and then job out the operating system development to a set of unrelated contractors. Each separate operating system can then be placed in long-term storage on the sensor platform, and a small boot system can be used to switch between the software. Since power is a restricted commodity on sensor platforms, each node can carry an array of electronic sensors, with the majority of these transducers being deactivated. The assignment scheme can also be used for determining the set of transducers which should be activated at any time. This reapplication of the diversity assignment paradigm discussed in the dissertation would ultimately reduce the risk of countermeasures against the sensing hardware.

Bibliography

[1] R. Albert, H. Jeong, and A. L. Barabási. Error and Attack Tolerance of Complex Networks. *Nature*, 406:378–382, July 2000.

[2] S. Alexander. Defeating compiler-level buffer overflow protection. *;login:*, 30(3):59—71, June 2005.

[3] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 217–224. ACM Press, 2002.

[4] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of vulnerability: A case study analysis. *IEEE Computer*, 33:52–59, December 2000.

[5] Giorgio Ausiello, M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi, and V. Kann. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag New York, Inc., 1999.

[6] A. Avižienis. Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing. In *Proceedings of the international conference on Reliable software*, pages 458–464, Los Angeles, California, 1975. ACM Press.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[8] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 281–289. ACM Press, 2003.

[9] S. M. Bellovin. Distributed firewalls. *;login:*, pages 39–47, November 1999.

[10] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington D.C., USA, August 2003.

[11] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack Magazine*, 0xA(0x38), May 2000.

[12] CERT Coordination Center. Incident and vulnerability trends. Technical report, CERT, May 2003.

[13] Z. Chen, L. Gao, and K. Kwiat. Modeling the spread of active worms. In *Twenty-Second Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, volume 3, pages 1890–1900, March – April 2003.

[14] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security; Repelling the Wily Hacker*. Addison-Wesley, Reading, MA, 2003.

[15] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 28–38, Chicago, IL, May 1998.

[16] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. Formatguard: Automatic protection from `printf` format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington D.C., USA, August 2001. USENIX Association.

[17] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard$^{\text{TM}}$: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, Aug 2003.

[18] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman. Raceguard: Kernel protection from temporary file race vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington D.C., USA, August 2001. USENIX Association.

[19] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*. USENIX Association, January 1998.

[20] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Foundations of Intrusion Tolerant Systems (OASIS'03)*, December 2003.

[21] L. J. Cowen, W. Goddard, and C. E. Jesurum. Coloring with defect. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 548–557. Society for Industrial and Applied Mathematics, 1997.

[22] J. Dadzie. Understanding software patching. *Queue*, pages 24–30, March 2005.

[23] C. Dahn and S. Mancoridis. Using program transformation to secure c programs against buffer overflows. In *10th Working Conference on Reverse Engineering*, pages 323–333, November 2003.

[24] T. de Raadt. Exploit mitigation techniques. In *The AUUG'2004 Annual Conference*, September 2004.

[25] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, February 1987.

[26] Y. Deswarte, K. Kanoun, and J. C. Laprie. Diversity against accidental and deliberate faults. In *Computer, Security, Dependability and Assurance (CSDA '98)*, pages 171–181, July 1998.

[27] Z. Dezső and A.-L. Barabási. Halting viruses in scale-free networks. *Physical Review E*, 65(055103), 2002.

[28] H. Ebel, L.-I. Mielsch, and S. Bornholdt. Scale-free topology of e-mail networks. *Physical Review E*, 66(035103(R)), 2002.

[29] J. Erickson. *Hacking: the art of exploitation*. No Starch Press, San Francisco, 2003.

[30] D. Estrin, L. Girod, G. Pottie, and M. Srivastava. Instrumenting the world with wireless sensor networks. In *Proc. International Conference on Acoustics, Speech, and Signal Processing*, Salt Lake City, Utah, May 2001.

[31] H. Etoh. GCC extension for protecting applications from stack-smashing attacks, 2004. Accessed on August 7th, 2005: http://www.trl.ibm.com/projects/security/ssp/.

[32] D. Evans. What biology can (and can't) teach us about security, August 2004.

[33] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 251–262. ACM Press, 1999.

[34] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 67–72. IEEE Computer Society, 1997.

[35] D. Geer. Monopoly considered harmful. *IEEE Security & Privacy Magazine*, 1(6):14–16, December 2003.

[36] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. P. Pfleeger, J. S. Quarterman, and B. Schneier. Cyberinsecurity: The cost of monopoly. Technical report, CCIA, 2003. Accessed on August 7th, 2005: http://www.ccianet.org/papers/cyberinsecurity.pdf.

[37] S. George, D. Evans, and L. Davidson. A biologically inspired programming model for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 102–104. ACM Press, 2002.

[38] S. W. Golomb. *Shift Register Sequences*. Holden-Day, Inc., San Francisco, CA, 1967.

[39] G. Goth. Addressing the monoculture. *IEEE Security & Privacy Magazine*, 1(6):8–10, December 2003.

[40] A. Gudmundsson and E. Chien. Security response: W32.klez.a@mm. Technical report, Symantec, 2001. Accessed on August 7th, 2005: http://securityresponse.symantec.com/avcenter/venc/data/w32.klez.a@mm.html.

[41] J. S. Havrilla and S. V. Hernan. Advisory CA-2001-06: Automatic execution of embedded mime types. Technical report, CERT, 2001. Accessed on August 7th, 2005: http://www.cert.org/advisories/CA-2001-06.html.

[42] T. G. Horsfall. *Genetic vulnerability of major crops*. National Academy of Sciences, 1972.

[43] A. Householder and R. Danyliw. Advisory CA-2003-08: Increased activity targeting windows shares. Technical report, CERT, 2003. Accessed on August 7th, 2005: http://www.cert.org/advisories/CA-2003-08.html.

[44] N. Ierace, C. Urrutia, and R. Bassett. Intrusion prevention systems. *Ubiquity*, 6(19):2–2, 2005.

[45] S. Jha, O. Sheyner, and J. Wing. Two formal analyses of attack graphs. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, page 49. IEEE Computer Society, 2002.

[46] M. K. Joseph and A. Avižienis. A fault tolerance approach to computer viruses. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 52–58. IEEE Computer Society Press, April 1988.

[47] J. E. Just and M. Cornwell. Review and analysis of synthetic diversity for breaking monocultures. In *Proceedings of the 2nd Workshop on Rapid Malcode (WORM)*, Washington, D.C., October 2004.

[48] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 272–280. ACM Press, 2003.

[49] J. O. Kephart and S. R. White. Directed-graph epidemiological models of computer viruses. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1991.

[50] A. D. Keromytis and V. Prevelakis. Dealing with system monocultures. In *NATO Information Systems Technology (IST) Panel Symposium on Adaptive Defense in Unclassified Networks*, Toulouse, France, April 2004.

[51] D. M. Kienzle and M. C. Elder. Recent worms: a survey and trends. In *WORM '03: Proceedings of the 2003 ACM workshop on Rapid Malcode*, pages 1–10, New York, NY, USA, 2003. ACM Press.

[52] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *Software Engineering*, 12(1):96–109, 1986.

[53] Marek Kubale. *Graph Colorings*, chapter Harmonious Colorings of Graphs, pages 95–104. American Mathematical Society, 2004.

[54] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws, with examples. *Computing Surveys*, 26(3):211–254, September 1994.

[55] J. P. Lanza and S. V. Hernan. Advisory CA-2003-07: Remote buffer overflow in sendmail. Technical report, CERT, 2003. Accessed on August 7th, 2005: http://www.cert.org/advisories/CA-2003-07.html.

[56] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington D.C., USA, August 2001. USENIX Association.

[57] R. A. Martin. Managing vulnerabilities in networked systems. *Computer*, 34(11):32–38, 2001.

[58] G. McGraw. Testing for security during development: why we should scrap penetrate-and-patch. *IEEE Aerospace and Electronic Systems Magazine*, 13(4):13–15, April 1998.

[59] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. Brite: An approach to universal topology generation. In *Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'01)*, page 346. IEEE Computer Society, 2001.

[60] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Network telescopes: Technical report. Technical Report tr-2004-04, Cooperative Association for Internet Data Analysis - CAIDA, 2004.

[61] D. Moore, C. Shannon, G.M. Voelker, and S. Savage. Internet quarantine: Requirements for containing self-propagating code. In *Twenty-Second Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, pages 1901–1910, March – April 2003.

[62] M. E. J. Newman, S. Forrest, and J. Balthrop. Email networks and the spread of computer viruses. *Physical Review E*, 66(035101), 2002.

[63] A. J. O'Donnell and H. Sethu. On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM conference on Computer and Communications Security*, pages 121–131, Washington, D.C., October 2004.

[64] A. J. O'Donnell and H. Sethu. Software diversity as a defense against viral propagation: Models and simulations. In *Symposium on Measurement, Modeling, and Simulation of Malware*, Monterey, CA, June 2005.

[65] R. Pastor-Satorras and A. Vespignani. Epidemic dynamics and endemic states in complex networks. *Physical Review E*, 63(066117), 2001.

[66] R. Pastor-Satorras and A. Vespignani. Epidemics and immunization in scale-free networks. In S. Bornholdt and H. G. Schuster, editors, *Handbook of Graphs and Networks: From the Genome to the Internet*, pages 113–132. Wiley-VCH, May 2002.

[67] R. Pastor-Satorras and A. Vespignani. Immunization of complex networks. *Physical Review E*, 65(036104), 2002.

[68] PaX Project. Address space layout randomization, Mar 2003. Accessed on August 7th, 2005: http://pax.grsecurity.net/docs/aslr.txt.

[69] C. Phillips and L. Painton Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 workshop on New security paradigms*, pages 71–79. ACM Press, 1998.

[70] A. Powell. Internet worms. Technical Report 00727, NISCC, 2003. Accessed on August 7th, 2005: http://www.niscc.gov.uk/niscc/docs/re-20030805-00727.pdf.

[71] N. Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, pages 1–14, San Diego, CA, August 2004. USENIX Association.

[72] D. Ressman and J. Valdés. Use of cfengine for automated, multi-platform software and patch distribution. In *Proceedings of the 14th Systems Administration Conference*, pages 207–218. USENIX Association, December 2000.

[73] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and Jr. W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 303–316. USENIX Association, December 2004.

[74] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th Systems Administration Conference (LISA)*, 1999.

[75] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, 2004.

[76] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address space randomization. In *Proceedings of the 11th ACM conference on Computer and Communications Security*, pages 298–307, Washington, D.C., October 2004.

[77] N. Sovarel, D. Evans, and N. Paul. Where's the FEEB? The Effectiveness of Instruction Set Randomization. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, 2005.

[78] M. Stamp. Risks of monoculture. *Commun. ACM*, 47(3):120, 2004.

[79] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, August 2002.

[80] Kyung suk Lhee and Steve J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–88, Berkeley, CA, USA, August 2002. USENIX Association, USENIX Association.

[81] G. Taleck. Ambiguity resolution via passive os fingerprinting. In G. Vigna, C. Kruegel, and E. Jonsson, editors, *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection*, pages 192–206, Pittsburg, PA, September 2003.

[82] The Honeynet Project. *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. Addison-Wesley, Boston, MA, 2002.

[83] J. Viega and G. McGraw. *Building Secure Software: how to avoid security problems the right way*. Addison-Wesley, Boston, 2002.

[84] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlang New York, Inc., 2001.

[85] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Proceedings of the International Conferece on Dependable Systems and Networks*, pages 193–202, Goteborg, Sweden, July 2001.

[86] Y. Wang, D. Chakrabarti, C. Wang, and C. Faloutsos. Epidemic spreading in real networks: An eigenvalue viewpoint. In *22nd Symposium on Reliable Distributed Systems*. IEEE Computer Society, October 2003.

[87] Y. Wang and C. Wang. Modeling the effects of timing parameters on virus propagation. In *Proceedings of the 2003 ACM workshop on Rapid Malcode*, pages 61–66. ACM Press, 2003.

[88] J. Wilander and M. Kamkar. A comparison of publically available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, 2003.

[89] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *Proceedings of the 18th Annual Computer Security Applications Conference*, page 61. IEEE Computer Society, 2002.

[90] G. Wroblewski. General method of program code obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP) 2002*, Las Vegas, USA, June 2002.

[91] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, pages 260–269, October 2003.

[92] J. Yan, A. Blackwell, R. Anderson, and A. Grant. Password memorability and security: Empirical results. *IEEE Security and Privacy*, 2(5):25 – 31, September – October 2004.

[93] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *IEEE Infocom*, volume 2, pages 594–602, San Francisco, CA, March 1996. IEEE.

[94] Y. Zhang, H. Vin, L. Alvisi, W. Lee, and S. K. Dao. Heterogeneous networking: a new survivability paradigm. In *Proceedings of the 2001 workshop on New security paradigms*, pages 33–39. ACM Press, 2001.

Appendix A. Host-Centric Diversity Assignments

In Chapter 3, we present a series of diversity assignment problems. For a single server, we are interested in maximizing the workload for an attacker who wants to mutate a commonly available exploit to be effective against our software installation. If we consider the hypergraph generated by our available diversity techniques, we can maximize the attacker workload by finding a path of hyperedges whose total effectiveness probability is minimized and whose total cost is below the cost bound. In terms of diversity techniques, we need to determine the subset of diversity techniques which should be applied to a single host that would maximize the resistance of a host to a known form of attack, given a maximum acceptable performance hit associated with the subset of diversity techniques used.

For example, consider a system which employs the following three diversity techniques: hardware diversification, address space randomization, and instruction set randomization. Address space randomization and instruction set randomization techniques have been characterized for their cost of implementation and effectiveness against attack for both 32-bit and 64-bit architectures. The cost of diversifying a pool of hardware to include both 32 and 64 bit platforms is non-zero, and significant, as compared to the cost of implementing either of the two previously discussed techniques. In Table A.1, we provide an example breakdown of these costs.

We define the host-centric diversity assignment system model in Table A.2. The assignment problem can be formalized, using the same framework created in Chapter 3, as follows:

HOST-CENTRIC DIVERSITY ASSIGNMENT:

- INSTANCE: A set D, the functions ρ and κ, a vector κ', and the set $\tau(A)$ which forms

Table A.1: A example set of the cost and effectiveness functions associated with multiple diversity techniques.

	Instruction Set Diversity (32-bit)	Address Space Diversity (32-bit)	Instruction Set Diversity (64-bit)	Address Space Diversity (64-bit)
ρ	$1/2^{32}$	$1/2^{15}$	$1/2^{64}$	$1/2^{40}$
κ	>1	1	>1	1

the domain of ρ.

- SOLUTION: A set of diversity schemes D_{opt} whose total performance hit does not exceed a preset performance deadline κ':

$$\forall m \in M\colon \prod_{d \in D_{opt}} \kappa(d,m) \leq \kappa'(m)$$

- MEASURE: The total probability of effectiveness of attack against the host selecting the optimization schemes:

$$\min_{D_{opt} \subseteq D} \sum_{t \in \tau(A')} w_\rho(t)\rho_{sol}(t)$$

where

$$\rho_{sol}(t) = \prod_{d \in D_{opt}} \rho(d,t)$$

Theorem 4 *The* HOST DIVERSITY ASSIGNMENT *optimization problem is NP Hard.*

Proof: Given an instance of MAXIMUM KNAPSACK, compute the antilog of the sizes and profits of each element to be placed in the knapsack. The antilogs of the sizes and profits of knapsack elements can be used as single cost and attack probabilities κ and ρ for a HOST DIVERSITY ASSIGNMENT solver. If an optimal solution for an instance of HOST DIVERSITY ASSIGNMENT can be determined in polynomial time, then MAXIMUM KNAPSACK can be solved in similar time. Accordingly, HOST DIVERSITY ASSIGNMENT exists in the same complexity space as MAXIMUM KNAPSACK. ■

Table A.2: An examination of the Intelligent Attacker vs. a Single Host attack and defense model.

	Intelligent Attacker vs. Single Host
Attack Model	We assume the attacker possesses an attack technique effective against a common distribution of the software which is running on the host. The attacker can mutate the attack so that it is effective against a variety of diversity techniques.
Cost of Attack or Effectiveness of Diversity	The cost and effectiveness metrics associated with each diversity technique provide figures for both the negative and positive impacts associated with the implementation of diversity on a host or set of hosts.
Defense Model	The system engineer may place a high priority on defense against a variety of attack techniques, including denial of service and buffer overflow remote exploitation. The engineer is also saddled with performance deadlines which must be met for the system to meet predefined benchmarks.
Composition	All the diversity techniques included are composable with one another. The composition of a set diversity techniques increases the attack resistance of a software package by the product of the *effectiveness probability* metrics of each of the diversity techniques.

Theorem 4 implies that the general form HOST DIVERSITY ASSIGNMENT problem can be solved using a multiobjective multidimensional version of a MAXIMUM KNAPSACK solver and a transformation on the hyperedge weights and costs ρ and κ. The multiobjective optimization requirement can be converted to a single objective by using the attack relevance metric w_ρ. Likewise, the performance impacts κ and their bounds κ' can be converted into a single metric using w_κ which encompasses the economic cost associated with the diversity technique's development cost and system performance degradation. These simplifications allow us to employ a bounded-performance greedy algorithm adapted from [5] for generating a feasible solution to the problem.

Figure A.1 contains the pseudo-code of the greedy knapsack algorithm adapted to the host diversity assignment problem. We define $\rho_s(d)$, $\kappa_s(d)$ and κ'_s as the single metrics which encapsulates the diversification effectiveness, diversification cost and the total diversification cost limit, respectively. The algorithm attempts to minimize the product of the effectiveness probability ρ by maximizing the sum of logs of ρ multiplied by -1 while keeping the sum of the logs of κ below the log of κ'.

As stated in Chapter 3, the costs of diversity techniques fall into several narrow bands based upon the development stage at which they are implemented. This fact, coupled with the limited amount of data available on the effectiveness of diversity techniques against all attacks and the cost of non-compiler based diversity techniques all but eliminates the need for a host-level diversity assignment algorithm at this time. In the future, the explicit measurement of both cost and effectiveness parameters will necessitate the use of such an algorithm, however.

```
# Receive input and initialize variables to zero
**Input** $D, \rho, w_\rho, \kappa, \kappa', w_\kappa, \tau(A), M$ as defined in the problem instance.
**Initialize** $D_{feas} \leftarrow \{\}$, $D_{ans} \leftarrow \{\}$
**Initialize** $\rho_{ans} \leftarrow 1$, $\kappa_{feas} \leftarrow 0$

# Weight metrics to single metric case
**for** $m \in M$:
    $\kappa'_s \leftarrow \kappa'_s + w_\kappa(m) * \kappa'(m)$
**for** $d \in D$:
    **Initialize** $\rho_s(d) \leftarrow 0$, $\kappa_s(d) \leftarrow 0$, $\kappa'_s \leftarrow 0$
    **for** $t \in \tau(A)$:
        $\rho_s(d) \leftarrow \rho_s(d) + w_\rho(t) * \rho(d,t)$
    **for** $m \in M$:
        $\kappa_s(d) \leftarrow \kappa_s(d) + w_\kappa(m) * \kappa(d,m)$

# Select only feasible diversity techniques
    **if** $\rho_s(d) > 0$ **and** $\kappa_s(d) < \kappa'_s$:
        $r_s(d) \leftarrow -\log \rho_s(d) / \log \kappa_s(d)$
        $D_{feas} \leftarrow D_{feas} \cup d$

# Begin greedy algorithm
**Sort** $D_{feas}$ **by** $r_s(d)$ **in decreasing order**
**for** $d \in D_{feas}$:
    **if** $\kappa' - \kappa_f eas \geq r_s(d)$:
        $D_{ans} \leftarrow D_{ans} \cup d$
        $\rho_{ans} \leftarrow \rho_{ans} \rho_s(d)$

# Check to see if a single diversity technique best
# satisfies the constraints, as opposed to a
# collection of diversity techniques
$\rho_{min} \leftarrow \arg\min\{\rho_s(d)\}$ as a function of $D_{feas}$
**if** $\rho_{min} \leq \rho_a ns$:
    $D_{ans} \leftarrow d$
**else**:
    **Sort** $D_{ans}$ **in order of temporal precedence**
**return** $D_{ans}$
```

Figure A.1: Pseudocode implementation of the Greedy Host Diversity Assignment algorithm.

Vita

Adam J. O'Donnell was born in Philadelphia at the Hospital of the University of Pennsylvania, not far from Drexel University. He graduated from Drexel Summa Cum Laude with a BSEE in 2001 and an MSCE in 2004. In a former life, Adam designed RF Amplifier subsystems at Lucent Technologies, where he was awarded a patent for his work. More recent times have found him consulting for many members of the computer security industry. Adam has worked on several books, serving as the technical editor and contributor to "Building Open Source Network Security Tools", a contributing author on "Hacker's Challenge", and co-author of "Hacker's Challenge 2".

During his time as a graduate student at Drexel, Adam investigated problems related to network economics, the topology of the Internet, and computer security. The work culminated in several research papers, including *On Achieving Software Diversity for Improved Network Security using Distributed Coloring Algorithms*, which first presented the ideas described in this dissertation. Adam's academic research was funded primarily by the National Science Foundation's Graduate Research Fellowship and through a collection of scholarships and grants, including the Koerner Family Fellowship, the Cisco Systems Information Assurance Scholarship, and the Colehower Fellowship.

Adam is a member of the IEEE, the ACM, the USENIX Association, and the Cult of the Dead Cow.