A Comparative Analysis of Denial of Service Vulnerabilities in Network Protocols

by

Samuel DeLaughter

B.A., Hampshire College (2012)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2019

© Massachusetts Institute of Technology 2019. All rights reserved.
A Comparative Analysis of Denial of Service Vulnerabilities in Network Protocols

by

Samuel DeLaughter

Submitted to the Department of Electrical Engineering and Computer Science on January 31, 2019, in partial fulfillment of the requirements for the degree of
Master of Science

Abstract

As global reliance on the Internet continues to grow, Denial of Service [DoS] attacks pose an ever greater threat to society. In recent years the proliferation of poorly secured "Internet of Things" [IoT] devices has enabled attackers to build massive botnets which have delivered on this threat with floods of unprecedented scale. Most mitigation tactics focus on re-routing this traffic or preventing devices from becoming compromised. While valiant, these efforts address only the symptoms of a deeper problem: design flaws in network protocols which make the servers supporting them vulnerable to attack. The tremendous complexity and diversity of protocols makes it difficult to analyze their security in a systematic way, and protocol designers still lack a set of best practices for DoS prevention. Establishing such guidelines will require the quantification of attack vectors to determine which protocol features tend to engender the most severe vulnerabilities. Yet an attack’s impact depends on characteristics of the server being targeted and of the services it offers, making objective measurement infeasible. Thus, we take a comparative approach to protocol analysis – tracking an array of server- and client-side metrics for various attack vectors in a controlled testbed environment. We develop a software suite for this purpose which can be easily extended to include new metrics and attack types. Initial data from this platform reveals novel attack vectors, indicates which existing protocols are most vulnerable, and lays a foundation for the systematic evaluation of new protocol specifications.

Thesis Supervisor: Karen Sollins
Title: Principal Research Scientist of the Computer Science & Artificial Intelligence Laboratory
A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.

- Douglas Adams
Acknowledgments

First and foremost, I would like to thank my advisor, Karen Sollins, for the tremendous guidance and support she has provided – not only on this thesis but throughout my two and a half years at MIT. She has encouraged me to pursue every question I’ve come across to its end, always seeing unexpected results as opportunities to learn something new. I could not have asked for a better mentor.

This work would never have been completed without the insight and expertise of Steve Bauer, who has a knack for asking the least obvious but most important questions, and who has helped me diagnose more problems than I would care to admit.

I thank Jon Proulx for keeping the infrastructure that supported this research up and running, and for the countless hours he spent debugging issues that likely only impacted my work.

Thanks also to Jim Kurose and Arun Venkataramani, with whom I had the pleasure of working before coming to MIT. They provided me with my first real research experience and sparked my interest in the field of Computer Networks.

Thanks to my officemates in the ANA group and other friends at MIT for being excellent sounding boards for my ideas, and for occasionally providing a much needed distraction from research.

I am eternally and indescribably grateful to my parents for the boundless love and support they have given me throughout my life. They are the kindest and most generous people I know; I aspire to live up to the example they set.

Finally, I would like to thank the person who encouraged me to apply to MIT, who constantly pushes me to seek more out of life and supports me tirelessly in that pursuit; my beloved partner and best friend, Kristína Moss Gunnarsdóttir. I would not be where I am or who I am without her.

This research was funded by the National Science Foundation
under Grant No. CNS-1413973
# Contents

1 Introduction 13

1.1 Motivation .................................................. 14
  1.1.1 The Threat of DoS ...................................... 14
  1.1.2 Shortcomings of Current Defenses ....................... 15
  1.1.3 Challenges of Secure Protocol Design ................... 17
1.2 Contributions ................................................ 17
1.3 Thesis Outline .............................................. 18

2 Background and Related Work 21

2.1 Designing a More Available Internet .......................... 21
  2.1.1 The Internet Protocol Stack ............................ 22
  2.1.2 How Protocols Are Made ................................ 23
2.2 Tools and Methods for Protocol Design and Evaluation ........... 26
  2.2.1 Taxonomies .............................................. 26
  2.2.2 Protocol Diagrams ..................................... 28
  2.2.3 Formal Languages ...................................... 28
  2.2.4 Formal Methods ....................................... 29
  2.2.5 Testbed Experimentation ............................... 30
  2.2.6 Network-Oriented Methods ............................. 31

3 Methodology 33

3.1 Experiment Structure ....................................... 33
3.2 Testbed Design ............................................. 34
3.2.1 Platform Selection ........................................... 34
3.2.2 Network Topology ........................................... 36
3.2.3 System Specifications ..................................... 37
3.3 Metrics .......................................................... 38
  3.3.1 Quality of Service ........................................ 38
  3.3.2 Resource Utilization ...................................... 39
3.4 Selected Attack Types ....................................... 41
3.5 Mitigations ..................................................... 45
3.6 Generating Attack Traffic ................................... 47
  3.6.1 Address Spoofing ......................................... 47
  3.6.2 Checksums ................................................. 49
  3.6.3 Rate Limiting .............................................. 50
  3.6.4 Randomness On-Demand ................................ 52

4 Experimental Results ........................................... 55
  4.1 Interpreting Plots ............................................ 56
  4.2 Quality of Service .......................................... 57
  4.3 Course-Grained Resource Utilization ...................... 58
  4.4 Fine-Grained Resource Utilization ....................... 60

5 Discussion ....................................................... 71
  5.1 Packet Length Guidelines .................................. 71
  5.2 Testbed Software ............................................ 72
    5.2.1 Code Structure ......................................... 72
    5.2.2 Data Structure .......................................... 73
    5.2.3 Testable Variables ..................................... 74
  5.3 Ethical Considerations ..................................... 75

6 Conclusion and Future Work ................................ 77

A SCTP Control Flow Diagrams ................................. 81
List of Figures

2-1 Formal Languages in RFCs ........................................... 29
3-1 Experiment Timeline .................................................. 34
3-2 Network Topology .................................................... 36
3-3 Comparison of TCP and SCTP Handshakes ......................... 44
3-4 The Long-INIT Attack ................................................ 46
4-1 Throughput ........................................................... 58
4-2 Jitter ................................................................. 59
4-3 Packet Loss .......................................................... 59
4-4 Round Trip Time ...................................................... 60
4-5 Network I/O (KB/s) ................................................... 60
4-6 Network I/O (packets/s) ............................................. 61
4-7 CPU Utilization ....................................................... 61
4-8 Memory Allocation ................................................... 61
4-9 Number of active 192-byte memory objects. ..................... 62
4-10 Total number of 192-byte memory objects. ..................... 62
4-11 Number of active 256-byte memory objects. .................... 62
4-12 Total number of 256-byte memory objects. ..................... 63
4-13 Number of active 512-byte memory objects. .................... 63
4-14 Total number of 512-byte memory objects. ..................... 63
4-15 Number of active 1024-byte memory objects. ................. 64
4-16 Total number of 1024-byte memory objects. ................. 64
4-17 Flame Graph of CPU Utilization During INIT Flood .......... 66
4-18 Flame Graph of CPU Utilization During INIT Flood (Zoomed) . . . 67
A-1 Legend . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
A-2 Core State Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . 83
A-3 Calculate RTO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
A-4 Compute Association Shared Key . . . . . . . . . . . . . . . . . . 85
A-5 Establish Association Shared Key . . . . . . . . . . . . . . . . . 86
A-6 Generate State Cookie . . . . . . . . . . . . . . . . . . . . . . . . . 87
A-7 Handle OOTB Packet . . . . . . . . . . . . . . . . . . . . . . . . . . 88
A-8 Handle Unexpected Cookie ECHO . . . . . . . . . . . . . . . . . . 89
A-9 Process ASCONF . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
A-10 Process DATA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
A-11 Process INIT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
A-12 Process SACK . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
A-13 Process ASCONF TLV . . . . . . . . . . . . . . . . . . . . . . . . 94
A-14 Send ASCONF . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
A-15 Send DATA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
A-16 Validate AUTH . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
A-17 Validate Cookie ECHO . . . . . . . . . . . . . . . . . . . . . . . . 97
A-18 Verify ABORT . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
Chapter 1

Introduction

The Internet has come to be a cornerstone of modern society, providing innumerable services that support everyday conveniences as well as critical infrastructure. As its value continues to grow, so does the incentive for malicious actors to disrupt it. One of the greatest security threats the Internet faces is that of Denial of Service (DoS) attacks, which attempt to make a website or other Internet-connected resource unavailable to its users. In recent years these attacks have become increasingly prevalent and powerful, and the security community still lacks a viable systematic approach to addressing the root cause of the problem: poorly designed network protocols.

Most DoS attacks are quite simple – they deliver a flood of network traffic to a targeted server in an attempt to exhaust one or more of its resources by brute force. If a server has to devote all of its computation, memory, or bandwidth to handling malicious packets, it loses the ability to provide an adequate quality of service to legitimate clients. The most common mitigation strategies for these attacks are to divert malicious packets away from their destination, or to prevent them from being sent in the first place. While these efforts may help reduce the frequency and potency of attacks, they are treating symptoms rather than the disease. To have any hope of ending the scourge of DoS attacks once and for all, we also need to design more resilient network protocols which prevent even the largest floods from exhausting a server’s resources.

Designing such secure protocols is challenging though – most are too complex for
formal verification to be a tenable option, and the community still lacks a comprehensive set of best practices. The best way to develop such guidelines is through the examination of known vulnerabilities. By understanding precisely which resources attackers target and how, we can uncover flaws in the designs of existing protocols. Understanding the commonalities between these flaws and evaluating the level of threat each of them pose will enable us to establish a set of principles for designing better protocols in the future.

In service of this goal, we develop a testbed environment for conducting controlled experiments to analyze and compare the impact of various DoS attacks. As a proof-of-concept, we conduct an initial set of experiments in this environment to analyze the relative effectiveness of several attack types, uncovering one novel vulnerability and gaining a better understanding of existing ones in the process. We have designed our testbed to be as flexible and extensible as possible, in hopes that other researchers will utilize it to expand upon our work, collaboratively constructing a database that encompasses as many different protocols, attacks, and target systems as possible.

1.1 Motivation

This research is motivated by a desire to improve the availability of services across the Internet. The threat posed by DoS attacks is substantial and ever-increasing, and the most common strategies for addressing the issue are insufficient and unsustainable. A more promising approach does exist, but there is a great deal of work that must be done to realize it.

1.1.1 The Threat of DoS

As global reliance on the Internet continues to grow, so does the threat posed by DoS attacks. These attacks are relatively simple to launch, made even simpler in recent years by the rise of “booters”, which offer DoS attacks against a target of choice at a specified time as a purchasable service.[40] This accessibility leads to a wide range of targets and perpetrators: a foreign government could impair our nation’s
critical infrastructure; politicians could silence their opponents or critical news outlets; investors could stall financial markets to manipulate stock prices; businesses could shut down their competitors’ online stores. Providers of DoS mitigation services have even been accused of launching attacks on potential customers to surreptitiously extort their business.[61]

In recent years, the advent of Internet of Things (IoT) technology has greatly exacerbated the problem, with billions of new Internet-connected devices flooding the consumer market. A lack of government regulation of or adequate frameworks for IoT security means that many of these devices are vulnerable to compromise by even the most basic means (e.g., devices shipping with default passwords like ”admin” that the user is not required to change, unnecessarily running insecure applications like Telnet [78], or transmitting sensitive information without the use of encryption).[33][45][55][60][64][104] Worse yet, it may be impossible for users to detect that such a device has been compromised, and even if they can many devices are incapable of receiving any security updates.[92] Malicious actors have seized on this bevy of susceptible gadgets to fuel DoS attacks of unprecedented scale, with the largest floods peaking at over a terabit per second.[57][59][101]

1.1.2 Shortcomings of Current Defenses

Brute force, flooding-based DoS attacks are particularly challenging to defend against. In order to provide a publicly available service on the Internet, a server must be willing to receive unsolicited traffic from unknown senders. Servers may attempt to detect which packets are malicious and which were sent by legitimate clients, but the two are often indiscernible from one another – any false positives which cause the server to drop good packets still constitute an effective denial of service to the clients that sent those packets. If attack traffic is sent from a single source address it can be blacklisted easily enough, but what about distributed attacks or those which utilize source address spoofing? If a compromised device happens to be a legitimate client of the service its botmaster uses it to attack, then its “good” and “bad” packets may be completely identical.
Even when traffic filtering works as intended, it still does not represent a real solution to the problem of DoS. It is merely a reactionary defense – if a server comes under attack these methods may allow it to maintain a better quality of service than it otherwise could, but they do nothing to prevent attacks from occurring. Thus, filtering only provides incentive to launch attacks which are larger, more widely distributed, and more difficult to detect.

The common preventative approach, bolstering device security to reduce the size of botnets, fails in much the same way. Security is an arms race, in which hackers seem capable of finding and exploiting vulnerabilities just as quickly as researchers and developers are able to patch them. New devices are constantly coming on the market, countless software updates are released every day, and there is no fool-proof way to verify their security. This gives our adversaries a clear upper-hand by providing them with a consistent supply of security flaws to take advantage of. As long as devices are capable of facilitating DoS attacks, botnet operators will continue finding ways to commandeer them. By fixing one vulnerability, we only incentivize them to find more. Ultimately, this approach may even drive bad actors to infiltrate device manufacturing companies and spur the production of intentionally insecure devices. Indeed, the Chinese government has already been accused\(^1\) of doing exactly that to some of the largest technology companies in the United States.\(^{[86]}\)

The most sustainable approach to stopping DoS attacks is by disincentivizing their perpetration. We must design network protocols in such a way that even the most clever attackers with the largest botnets at their disposal are incapable of significantly disrupting any online services. By minimizing the resources allocated for each individual packet, we can decimate the impact of even the most massive floods. Doing so would improve availability Internet-wide, reduce the need for active defense measures in the network, and eliminate a major incentive for attackers to take control of consumer devices.

All that being said, the protocol design-oriented approach is not a panacea, and

\(^1\)It should be noted that the veracity of this reporting has been disputed by some of the companies involved, and by the United States Department of Homeland Security.\(^{[22]}\)\(^{[23]}\)\(^{[91]}\) Yet whether or not this particular incident actually occurred, the threat of a similar compromise remains very real.
despite their shortcomings there is still tremendous value in continuing to deploy and improve other preventative and reactive mitigation strategies. While we are optimistic about our method’s potential, we also recognize that fully realizing our vision of designing and building an intrinsically DoS-resilient internet will take a great deal of time and effort. Even if and when that future does come, new devices will always be produced, protocols will always need to be updated, and mistakes will inevitably be made. In those circumstances, the strategies above will offer a necessary fallback defense.

1.1.3 Challenges of Secure Protocol Design

Though the optimal solution to the threat of DoS attacks is clear, the path to implementing it remains significantly less so. Evaluating the security of disparate network protocols in a systematic way is extremely challenging. While certain vulnerability types appear in multiple protocols, many are highly protocol-specific. Furthermore, protocol specifications receive updates on a fairly frequent basis which may introduce new vulnerabilities, even in the process of addressing existing ones. Even if the impact of a given update is purely positive, building confidence in that fact can require a great deal of time and effort. Further details on the protocol design process and the challenges therein is provided in Section 2.1.

1.2 Contributions

The primary contribution of this thesis is the design and development of a controlled test environment tailored specifically for the comparative analysis of DoS vulnerabilities in network protocols. It is designed to be as accessible, general, and extensible as possible. It may be run on any set of Linux devices, and provides modular support for adding new attack types, client applications, and metrics. Also included are tools for automating the collection, processing, and visualization of test data. The source code and documentation for this testbed are publicly available at [37].

Additionally, we deploy a version of our testbed and use it to evaluate several
different attack types. The results of these tests constitute this work’s second largest contribution. We identify and pinpoint the cause of one novel vulnerability in the Stream Control Transmission Protocol and gain deeper insight into the mechanism underlying a well-known vulnerability in the User Datagram Protocol. We provide what is to our knowledge the first empirical evaluation of the relationship between the length of packets in a flooding attack and the impact of that attack. Based on these results, we offer some simple suggestions as a first step towards crafting a comprehensive set of design principles to help protocol designers build a more DoS-resilient Internet. The raw data from our experiments is available at [38].

We also create a detailed set of diagrams illustrating the control flow of the Stream Control Transmission Protocol. These diagrams are somewhat tangential to the rest of our work, intended primarily to increase our own understanding of the protocol’s operation and to demonstrate the level of complexity involved in designing and specifying a feature-rich network protocol. Nonetheless, they represent a substantial amount of effort and help inform our discussion of protocol design practices.

1.3 Thesis Outline

The underlying ethos of this thesis is a desire to design and implement network protocols that are more resilient to DoS attacks, so that we may build from those protocols a more secure internet.

Chapter 2 discusses existing research related to these efforts and general principles which guide our work. We begin by describing exactly what it means for an internet to be secure, specifically in the context of availability. Next we discuss the unique challenges DoS attacks present and common methods for evaluating their impact. We then detail the current protocol design and specification process as well as the pros and cons of some potential tools and methods that could be used to improve it, most notable among them being the empirical measurement of client quality of service and server resource utilization in a controlled network testbed.

In Chapter 3 we lay out the design for such a testbed and the structure of experi-
ments to be conducted therein. We have built a version of this testbed, designed to be as easy as possible for the community to utilize and to extend with their own attack types, metrics, and network topologies. In it we’ve conducted a set of experiments to analyze if and how the length of packets sent in a flooding attack influences the quality of service provided and resources consumed by that attack’s victim(s). Our findings are presented in Chapter 4.

We then discuss guidelines for protocol design one could infer from our results, the operation and capabilities of our software, and the ethical considerations pertinent to this sort of research in Chapter 5. Finally, we conclude and consider possible future work on this topic in Chapter 6.
Chapter 2

Background and Related Work

Here we present a more detailed definition of the goals outlined in Chapter 1 and a discussion of prior work towards attaining them upon which this thesis builds. First we define what exactly an internet is and what it means for one to be secure. We then explain why availability is a particularly important aspect of security and why it is exceedingly difficult to guarantee. This includes a discussion of current procedures for the design, specification, and implementation of network protocols and a comparison of some potential improvements to those procedures, including the approach of empirical analysis for which we design, deploy, and test a set of tools in Chapters 3 and 4.

2.1 Designing a More Available Internet

This research is part of an endeavor to design and build a better internet. The lowercase ‘i’ in “internet” here is important – it denotes the general concept of an interconnected set of autonomous systems, capable of communicating via some set of standardized protocols. The capital-‘I’ Internet that most everyone uses is simply one possible implementation of this concept. It is by far the most successful to date, but that fact should not be misconstrued as proof that it is ideally designed or impervious to supersession. The Internet has been a phenomenally successful and fundamentally world-altering technology, but there is always room for improvement.
Indeed, efforts to rethink internet design are already well underway. The National Science Foundation’s Future Internet Architectures project (FIA) [73] (along with its precursor, the Future Internet Design initiative (FIND) [12]) has solicited and supported multiple proposals to design a new internet from the ground up, guided by lessons learned since the current Internet’s inception about its potential use cases and implicit limitations. Given the amount of sensitive and vital information transmitted over the Internet (financial transactions, medical records, government communications, etc.), security has been identified as one of the most vital characteristics for any successful proposal to possess. Yet “security” is an incredibly broad goal, encompassing a wide array of features including authentication, authorization, access control, data integrity, and non-repudiation.

One particularly important yet often overlooked aspect of security is availability, the ability to interact with services in a timely manner. If malicious actors are capable of preventing Internet users from accessing online services, then the Internet has ceased to provide interconnection between devices and has therefore failed by definition. Thus for a protocol to truly be secure, relying on it must not make a device or application unduly vulnerable to attacks which prevent it from providing or receiving an acceptable quality of service. So how should we go about crafting such protocols? This question demands an examination of our current Internet’s protocols, and the procedures involved in their creation.

2.1.1 The Internet Protocol Stack

The current Internet uses a layered suite of protocols. Its core language is the Internet Protocol (IP) as defined in [81], which constitutes the so-called “narrow waist” of the protocol stack; it may rest atop many different lower-layer protocols, and beneath

\[And\ \text{arguably IPv6 as defined in [36]. Migration from IPv4 to IPv6 has been ongoing for nearly a quarter century, since the new version was first standardized in 1995. [50] Asking whether the burgeoning IPv6-based Internet and the slowly fading IPv4 Internet are one and the same is more of a philosophical question than a technical one, evoking the “Ship of Theseus” thought experiment: if each plank of a boat’s hull is replaced one at a time over many years, is it still the same boat?}\]
many possible higher-layer ones.\textsuperscript{2} Layering protocols provides modularity. Only the most essential features are included in the stack’s narrow waist, while devices and applications may pick and choose the higher- and lower-layer protocols that provide the specific supplemental features they require.

The downside of layering is that it creates more complexity. Verifying the correctness of a two-protocol suite is at least twice as hard as verifying a single protocol; we must analyze not only the two individual protocols but the interaction between them as well. This complexity grows exponentially as more protocols are added, and the current Internet landscape contains hundreds of them. Ensuring a perfectly secure Internet would require verifying each one of them and all possible layered combinations. This is simply intractable with current methods. Thus, it is unlikely we will ever have an internet that is completely immune to DoS attacks and capable of providing unwavering availability, but there is much room for improvement beyond the current state of affairs.

\subsection*{2.1.2 How Protocols Are Made}

The standardization process for network protocols is vitally important. Interoperability requires consistency; if two entities disagree on any detail of some protocol they use to communicate with one another, that communication is likely to fail. Thus, it is valuable to examine the current procedures in place for designing, standardizing, and implementing network protocols.

The IETF

The organization responsible for designing and standardizing the vast majority of protocols in use on today’s Internet is the Internet Engineering Task Force (IETF). As such, any productive discourse on protocol design necessitates a basic understanding of this body’s structure and procedures. As its website states:

\footnote{Lower-layer protocols are those closer to and including the physical connections between devices, such as Ethernet and Wi-Fi[4]. Higher-layer protocols are those closer to and including interfaces with end-user applications, such as TCP[82], SSH[103], and SMTP[58].}
“The IETF is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet.”[1]

In essence, protocols are conceived and meticulously defined by whomever is interested in doing so. Companies that rely on IETF standards may pay employees to contribute time and labor, but all contributors are viewed by the IETF as individual volunteers, never as representatives of outside organizations.[13] Most work is conducted via publicly accessible mailing lists (with the exception of three in-person meetings per year), and decisions are made by “rough consensus.” It is rather remarkable how effective this system has been at producing and maintaining such a large set of standards upon which so much of modern society depends. Yet despite this tremendous success, major vulnerabilities do slip through the cracks from time to time.

Since 1997, all RFCs produced by the IETF have been required to “contain a section near the end of the document that discusses the security considerations of the protocol or procedures that are the main topic of the RFC.”[80] Citing the typical inadequacy of these Security Considerations sections, the best current practices for writing them were expounded by RFC 3552 in 2003.[85] This document does a good job of defining security objectives, describing common attack types, and offering examples of well-written RFCs, but the Internet landscape has changed dramatically over the 15+ years since it was written. Security analysis should be a fundamental step in the protocol design process, and to be effective it must be informed by a current set of best practices.

Worse yet, the IETF has thus far stopped short of mandating any specific type of analysis designers must perform or degree of detail they must provide in the Security Considerations section. A protocol standard may be approved without giving any consideration whatsoever to the notion of availability. Even if availability is discussed, there is no guarantee that all vulnerabilities will be mentioned. We would recommend that the IETF revisit its Security Considerations guidelines, update its set of examples, and consider imposing more stringent requirements. Specifically, we
suggest mandating some mention of DoS vulnerability (even if only to say that none has been found), and incentivizing the use of one or more tools described below in Section 2.2.

**Moving from Design to Specification to Implementation**

The IETF is not the only piece of the puzzle – once a protocol has been designed and standardized it must still be implemented. Ensuring the security of a protocol requires verification at each stage in this process and verification that the conversion from one stage to the next is correct. A provably secure specification is worthless if no one is able to implement it correctly, as is an implementation that conforms precisely to a flawed design. Since the implementation of IETF standards is carried out by third parties (primarily kernel and operating system developers), there is no single authority capable of overseeing and validating the full sequence of a protocol’s creation from conception to deployment. Instead, each step must be made as unambiguous as possible to facilitate its correct interpretation down the line.

Flaws in a design may result from mistakes made by its contributors or from intentional trade-offs intended to avoid worse problems. Such trade-offs are an integral part of the protocol design process – it does not make sense for every protocol to offer every conceivable feature, and indeed certain features are mutually exclusive. Some features must therefore be prioritized over others, based on assumptions made about the resources available to the devices expected to run a protocol. There is always a chance that a protocol’s real-world applications will diverge from its designers’ expectations, in which case those assumptions may not hold up. Additionally, the IETF’s policy of making decisions by rough consensus means that choices may favor the opinions of the most senior and/or vocal contributors, even if they happen to produce sub-optimal results.

Implementation flaws may be caused by careless errors in the code, or by ambiguous language in the specification which fails to convey an important design choice to whomever writes the code. There is a tension that exists in the design process between simplicity and clarity. A protocol’s specification must contain enough detail
to enable multiple programmers to independently develop interoperable implementations, but should also make it easy for a non-expert to understand the protocol’s basic operation.

2.2 Tools and Methods for

Protocol Design and Evaluation

We must first ask whether a protocol is designed to provide the highest level of security possible for the features it requires, then whether it is written clearly enough to facilitate interoperable implementations, and finally whether each implementation actually conforms to the specification. Current approaches to each of these questions lack systematicity, but there is a large body of prior work which offers potentially useful tools for improving this state of affairs.

2.2.1 Taxonomies

In attempting to approach protocol security more systematically, it is valuable to have some cohesive way of classifying and cataloging known vulnerabilities. This is true for our specific goal of improving availability as well; giving designers a list of common DoS attack types should help them avoid repeating the mistakes of the past.

The most comprehensive work on this topic to-date is the 2004 paper “A Taxonomy of DDoS Attack and DDoS Defense Mechanisms” by Mirkovic and Reiher.[69] The authors categorize attacks based on:

“...the means used to prepare and perform the attack (recruit, exploit and infect phases), the characteristics of the attack itself (use phase) and the effect it has on the victim.”

The cost to the attack’s victim is clearly of most relevance to our work, but only by considering this in context of the attack’s structure can we begin to recognize patterns over time. If for example we notice that the most damaging attacks all
require source address spoofing, we know that there is value in designing protocols to explicitly prevent such behavior.

For a taxonomy to be useful though, it must be kept up-to-date. In the 15 years since [69] was published, the IETF has released 4,766 new RFCs— that’s approximately 56 percent of all RFCs in existence.4[15] Not all of these specify protocol standards, but nonetheless this statistic makes the need for an updated taxonomy glaringly obvious. What insights can protocol designers be expected to draw from a taxonomy that was compiled before many modern protocols (and attacks targeting them) existed?

Granted, updating a taxonomy is not quite as straightforward as it sounds. It may be unclear how a newly discovered vulnerability fits into an existing framework, potentially requiring the addition of entirely new categories. Furthermore, relying on even an up-to-date taxonomy in analyzing a protocol’s security could prove problematic. Apparent commonalities between protocols may be misleading – just because two attacks target the same resource or leverage the same type of protocol feature does not necessarily mean the same mitigation strategy is appropriate for both of them. Researchers may also be lulled into a false sense of security if they simply assume a given taxonomy is complete and fail to search for vulnerabilities beyond its bounds.

The one area in which taxonomies truly shine is their construction of a common language for the research community. Even if we can’t provide a comprehensive list of all vulnerabilities, there is great value in the ability to communicate clearly about the ones we are aware of. Protocol specifications should take advantage of this, and refer to DoS vulnerabilities using terms from the most up-to-date and well-known taxonomies available.

---

3A Request For Comments (RFC) is the name given by the IETF to documents specifying protocol standards, best practices, and other related information.

4The rate of publication has increased dramatically since the first RFC was released in 1969.
2.2.2 Protocol Diagrams

Many specifications include a diagram of their protocol’s core state machine, showing the various states it may be in and the events which trigger transitions between them. Such visual aids can make it easier for non-experts to decipher a protocol’s basic operation, and also provide a template for coding implementations. However, it is rare to see more than the basic states of a protocol diagrammed – finer details are usually left in plain English. This is unfortunate, but understandable given the complexity of diagramming certain control structures. As an illustration of this potential complexity we have created diagrams representing nearly the entire control flow of SCTP, which are provided in Appendix A.

2.2.3 Formal Languages

One way to ensure precision in a specification is to write some or all of it in a formal/technical language. Describing systems in code (or at least pseudocode) rather than English removes ambiguity and gives implementers a much simpler starting point to work from. We would therefore expect to find far fewer implementation flaws in a protocol that contains formal language in its specification. One downside of this approach is that it may make the specification more difficult to understand for non-experts.

The most recent discussion of this topic we were able to find on the organization’s website is in a 2001 blog post, which states:

> Formal languages are useful tools for specifying parts of protocols. However, as of today, there exists no well-known language that is able to capture the full syntax and semantics of reasonably rich IETF protocols.

> We therefore expect that people will continue using English to describe protocols, with formal languages as a supporting mechanism. [51]

Nearly two decades later, this prediction appears accurate. While the specific languages that are in use have changed over time, there has not been a significant
uptick in the overall percentage of RFCs which use one or more formal languages. These trends are shown in figure 2-1, with data taken from [51].

![Percentage of RFCs Using Formal Languages](image.png)

Figure 2-1: The percentage of RFCs which utilize formal languages. Statistics on languages other than these six (ABNF[75], ASN.1[24], C[6], CBOR[28], JSON[30], and XML[3][87]) are not provided by the IETF. Note that a single protocol standard may make use of multiple different languages.

### 2.2.4 Formal Methods

Formal methods take formal languages a step farther. By specifying a system design in a particular way, researchers can test whether certain conditions hold across all possible executions of a protocol, and (if so) can state that the protocol is *provably secure* in the context of those conditions. The application of formal methods to the design of communications protocols dates back over forty years. Early efforts are surveyed in [27] and [96], and an overview of more recent work is provided in [98].

Given the power that provable security would bring to the design process and the amount of time the scientific community has had to develop the necessary tools, one may question why the IETF does not require a formal security analysis of all protocols before publishing their specifications. Unfortunately, as indicated in [98], the current state-of-the-art in formal analysis is still lacking in its ability to assert anything other than the logical correctness of a protocol. This is insufficient; in the context of avail-
ability we also need to be able to make guarantees about a protocol’s performance. Doing so requires some means of incorporating into the formal specification notions of time and distributed state, which is not possible with most formalization techniques.

Model checkers that do provide ways of tracking state would suffer from state explosion, as they must search all possible states of all connected devices to definitively rule out any and all vulnerabilities – the Internet’s massive scale and heterogeneity clearly make this intractable. A limited, probabilistic search of the state space would be feasible but insufficient. DoS attacks often leverage corner cases; a protocol may be vulnerable only to one very particular packet structure which a probabilistic approach could easily fail to check.

While it remains a promising field of study, we cannot expect the formal analysis of network protocols to become standard practice anytime in the near future. Protocol designers and implementers should be incentivized to use it whenever practical, but mandating its use would only serve to slow the protocol development process and stifle innovation.

2.2.5 Testbed Experimentation

Even if a protocol has been “proven” secure through static analysis, there is always a chance that this proof was structured incorrectly or failed to take into account some important condition. Thus, we need a more objective method for assessing a protocol’s vulnerability to attack. One solution is to simply observe the protocol in action and measure its performance while under attack. This can enable us to find vulnerabilities that a formal proof may miss, and to compare the relative security of different protocols or different implementations of the same protocol.

One downside of this approach is that it requires more back-and-forth between designers and implementers – potential designs should be tested before publishing a specification, but testers must have some form of specification to work from when building their implementation. However, the IETF does require at least one test implementation to be built before a protocol standard is published.[16] They could require test implementations to meet certain security benchmarks before approval, as
established by results from similar existing protocols. Additionally, a new developer could compare her implementation with the results of others to determine whether it provides a reasonable level of security.

The other issue is that experiments must be performed in similar (or preferably identical) environments to provide comparable results. The community still lacks a standard test environment that is accessible to all interested parties and flexible enough to test the full spectrum of attacks against any given protocol. The bulk of our work is devoted to meeting this need. In chapters 3 and 4, we develop a testbed environment that is intended to be as inexpensive, versatile, and extensible as possible, and design and conduct some experiments to illustrate its utility.

These efforts build on a large body of prior work on the empirical evaluation of DoS attacks. Chertov et al. explore the trade-offs of testing in emulation versus simulation in [32]. Kaur et al. compare various tools commonly used to launch real-world DoS attacks in [56]. Bhuyan et al. provide metrics for characterizing attack traffic in [26]. Empirical measurements of specific attack types in test environments are performed in [84] [88] and [93], and assessments of past real-world attacks are provided in [49] [62] and [100]. Most notable is the work of Mirkovic et al. who have a number of papers which heavily inform our work. They establish client- and server-side metrics, benchmarks, and testing methodologies in [66][67][68] and [70], and describe a user-friendly automated test environment in [71]. Our decision of which metrics to implement was based primarily on findings in [67], and our desire to design and build our own testbed was spurred by the discovery that the tools they developed are no longer maintained or readily available to the public.

2.2.6 Network-Oriented Methods

The tools and methods above pertain mainly to securing endpoints by designing better protocols for them to run. An alternative approach to improving Internet-wide availability is through the design of network infrastructure between hosts. To

\footnote{It should be noted that our attempt to reproduce the work of Rathgeb et al. in [84], an evaluation of DoS vulnerabilities in SCTP, is what originally sparked this research.}
this effect, Yang et al. present a “DoS-limiting Network Architecture” called TVA in [102]. Their architecture requires that hosts request and obtain permission prior to sending packets to one another, with in-network logic working to verify that each packet contains the requisite proof of this permission. While this architecture may be effective at DoS mitigation, it would require more network overhead than IP, and convincing a critical mass of Internet users to migrate to an entirely different architecture is a daunting task (as evidenced by the slow transition from IPv4 to IPv6 and the amount of time and effort that has been invested in the FIA project [73]).

Smith and Schucard propose using reactive BGP routing to mitigate DoS attacks, by sending traffic over links that are not congested with malicious traffic.[94] This approach seems sensible when bandwidth is the primary resource being targeted by an attack, but is not applicable to low-rate attacks that target other system resources.

Along the same lines, there is value in the ability to assess the overall health of the Internet, rather than just that of a single host or connection between hosts. Moore et al. present methods for measuring Internet-wide DoS activity in [72], and Beverly et al. evaluate the deployment status of ingress filtering to defend against address spoofing attacks in [25]. The detection and measurement of botnets is covered in [76] and [100]. While these works do not directly advance efforts to build a more DoS-resistant Internet, they do give us valuable tools for evaluating our progress along that path.
Chapter 3

Methodology

The primary goal of our experiments is to provide a controlled, reproducible comparison of different DoS attack variants in terms of their potential to waste a targeted server’s resources and degrade or disrupt its client connections. This approach allows us to push networked systems into scenarios that are highly unlikely to occur naturally but may nevertheless be leveraged by a prescient attacker. To that end, we have developed an extensible set of software utilities for executing, measuring, and analyzing arbitrary denial of service attacks. This chapter outlines the structure of a generic experimental run in our environment, design decisions that went into building our testbed, the various metrics we track, and the attack types we have chosen to implement thus far.

3.1 Experiment Structure

Each experiment measures the impact of a particular DoS attack on an ongoing client-server connection. First, the client and server begin an association. Once baseline measurements have been established, a set of attackers begin to flood the server with traffic. After some time, the server implements a mitigation against this attack. Throughout this process, measurements are taken at the client to quantify the quality of service (QoS) it is receiving, and at the server to determine which resource(s) (broadly: bandwidth, computation, and/or memory) the attack is preying upon.
Figure 3-1 illustrates the general trends in both QoS and resource utilization that we expect to see over the course of an experiment. If the attack being tested is viable, QoS will drop when it begins, while the utilization of one or more resources increases as the server struggles to process the incoming flood. Once a successful mitigation is implemented, resource utilization recedes as malicious packets are detected and discarded, and QoS is restored (though it may remain below its pre-attack level).

<table>
<thead>
<tr>
<th>Time</th>
<th>Quality of Service</th>
<th>Resource Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack Begins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mitigation Implemented</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-1: The general timeline of a single experiment, and expected trends in QoS and resource utilization as it progresses.

### 3.2 Testbed Design

Here we present the design decisions made in building our test environment. This includes the hardware we use, the topology of our experimental network, and the kernel and operating system running on our hosts.

#### 3.2.1 Platform Selection

We initially began conducting experiments in Mininet, a freely available network virtualization software with many compelling properties.[10] It enables the emulation of a multi-device network within a single virtual machine (VM), providing a great deal of flexibility to customize the network topology and a simple commandline-interface
for communicating with hosts. Mininet lends itself well to conducting reproducible research, as a VM is entirely self-contained and easy to share. The VM environment is also well-isolated from outside influence, but does not provide the most realistic message-passing model or the most accurate means of measuring server resource utilization. Additionally, we discovered that modifying certain kernel-level parameters on the virtual hosts was impossible. This lack of accuracy and control forced us to find an alternative.

We considered setting up a physical network testbed, as this would provide unparalleled accuracy in our measurements, total control over all system parameters, and effectively perfect isolation (as it could be completely disconnected from the Internet). Unfortunately, physical servers are expensive to acquire and maintain. They also make it difficult to rapidly implement topology changes, or to scale up the network if additional hosts are needed to strengthen an attack. Given that we had limited funding and did not know in advance how many hosts our experiments would require, we determined that building a physical testbed would be impractical.

Ultimately, we settled on using the CSAIL OpenStack cluster [34][74], a cloud computing environment similar to Amazon Web Services [2], Microsoft Azure [9], or the Google Cloud Platform [5]. It allows us to set up a discrete VM for each host and a dedicated private network to connect them all. Hosts can be added, removed, reimaged, or reprovisioned on the fly. The private network is well isolated from the outside world, and we have complete control to update and modify the kernel and operating system of every host.

Our VMs do share some resources with those of other users which prevents us from measuring as accurately as we could in a dedicated physical testbed, but this model is still much more realistic than virtualization. In order to reduce potential noise from outside activity, we repeat each experiment multiple times and average the results. Our cluster is generally very under-loaded so this repetition should suffice to dampen the impact of any rare events in which our hosts have to compete for resources. Our objective is not to provide an objective measurement of any one attack’s impact, only to compare the relative impact of different attack types.
We deploy each of our hosts with the same image – this will make our work easily reproducible as others may copy and use our image to deploy their own virtual hosts on OpenStack or a similar platform. It may even be deployed on a physical testbed for researchers who have one available.

3.2.2 Network Topology

Our topology includes a client, a server, twenty-five attackers, and a “bridge” host. The bridge serves as a buffer between our network and the Internet, and as the point from which scripts on the other hosts are launched via SSH commands. It is impossible to connect to any of the other hosts from the public Internet without establishing an SSH tunnel through the bridge. This isolation is important because it keeps stray packets from reaching our network and interfering with our experiments, and also prevents our attack traffic from leaking out and accidentally harming real-world systems.

The physical interconnection between hosts on our network is handled by OpenStack, the network links of which have more than sufficient capacity to prevent them from becoming a bottleneck. A figure illustrating our topology is provided in Figure 3-2.

![Network Topology Diagram]

Figure 3-2: The network topology for our experiments.
3.2.3 System Specifications

All hosts run Ubuntu Server 18.04.1 LTS [17] with Linux kernel version 4.15.0-22-generic [18]. Of the top 10 million websites on the public Internet, 69% are Unix-based and at least 53.8% of those run Linux\(^1\), giving it a larger install-base than any other kernel for the type of systems we are most concerned with.[20][21] Furthermore, Ubuntu is the single most common flavor of Linux, accounting for 38.1% of Linux servers measured.[19] Linux-based operating systems also work more easily with OpenStack than any others. While we have tried to select the most representative single operating system possible, future development of our software would benefit greatly from adding support for BSD, Windows, and others. Running tests on multiple different operating systems would allow us to determine whether a given vulnerability is implementation-specific or represents a real design flaw in a protocol.

The particular kernel version we tested on was simply the most recent stable release available at the time we selected it. Older kernel versions are likely to lack some useful system profiling tools, and their implementations of the network protocols we examine may be buggy or poorly optimized. Indeed, we conducted earlier experiments with an older version of the Linux kernel and found our server to be significantly more resilient to attack once we updated it. Using the latest stable kernel version available is a good way to guarantee that results more accurately reflect the behavior of up-to-date systems. One should avoid combining data from experiments executed on different kernels.

The server and client each have a single virtual CPU (VCPU) and 512MB of RAM. This is modest by modern standards, but with good reason. Analyzing CPU utilization would be considerably more complex on multi-core systems. Also, as we lack the resources to launch very large-scale floods, a better-provisioned system would likely show little or no sign of duress while undergoing our attacks. Attackers also have a single VCPU, but are equipped with 2GB of RAM. We found this necessary for sending traffic at a high rate, as attack packets must be queued in advance (discussed

\(^1\)This portion is more substantial than it seems, as 45.3% are running an "unknown" Unix variant. The second most popular variety is BSD, which accounts for just 0.9% of public Unix systems.
in further detail in Section 3.6). We take care not to run or even install extraneous applications on any host to avoid interfering with our experiments.

### 3.3 Metrics

This section details the specific metrics we track, the utilities we use to do so, and our reasons for selecting each. We have designed our test environment and data analysis tools to be modular, such that they can easily include additional server-, client-, and attacker-side metrics as needed. Our goal is not to show whether any one attack type crosses a meaningful threshold for some particular metric, but to compare how an array of different attacks compare across all metrics. From those general observations, the reader (or a user of our software) may determine for herself whether a given protocol is a safe choice for the type of application she intends to build, based on the communication paradigm it requires and the amount of each resource her server has available. We divide our set of metrics (and the remainder of this section) into two categories: Quality of Service, and Resource Utilization.

#### 3.3.1 Quality of Service

Client-side measurements are the most direct way to evaluate how a DoS attack impacts a user’s experience when they attempt to utilize the targeted service. In selecting the initial set of metrics to implement, we turned to existing work on quantifying QoS, most notably that of Mirkovic et. al.. [68] The overall most valuable metrics they identify are: one-way delay, round trip time (RTT), packet loss, transaction duration, throughput, and jitter. Yet as their results show, the relative value of these is highly application-specific. For example, packet loss has a large effect on the quality of streaming audio and video, while transaction duration is more meaningful for HTTP(S) and FTP sessions.

We measure throughput, jitter, and packet loss using iPerf version 3.1.3.\(^2\).[41] The

\(^2\)We chose to use this specific version of iPerf because it supports sending client traffic over the SCTP protocol in addition to UDP and multiple variants of TCP.
client initiates a connection to the server which then sends a continuous 100Mbps³
stream of UDP data back to the client. This is intended to approximate a streaming
audio/video or FTP-esque application, in which the user requests a large piece of
content that the server then delivers in segments.

In separate runs, we measure RTT using the ping command. The client sends
a 64-byte ICMP Echo Request [79] to the server twice per second and measures the
time it takes to receive a response. Measuring one-way delay would require clock
synchronization between hosts, which we do not implement in order to avoid the
extraneous traffic of a protocol like NTP [31] interfering with our measurements.
Given the simplicity of our network topology, it is reasonable to assume that the
one-way delay will be half of the RTT on average.

We have not implemented a metric for total transaction duration, as that depends
on the amount of information a transaction contains and the amount of client-server
interaction it requires to complete, and is therefore highly application-specific. Pro-
vided with those details one could infer an approximate transaction duration from
throughput and RTT results, or may extend our software to include a direct metric
for their own particular transaction type(s).

### 3.3.2 Resource Utilization

While not particularly reliable indicators of client QoS, server-side measurements
provide vital information when attempting to discern the root cause of a DoS vulner-
ability. If the increased consumption of a given resource correlates with a drop in QoS
during an attack, that attack is likely leveraging a design flaw or implementation bug
that enables unnecessary waste of that resource. The essential server resources we
monitor are CPU utilization, memory allocation, and network bandwidth consump-
tion.

³We also rate-limit the server’s outgoing bandwidth to 100Mbps using the tc utility. Setting
these values equal to one another improves the visibility of our attacks, which is necessary as we
do not have a large number of attackers at our disposal. The relative strength of different attack
types looks the same when the server has extra bandwidth, though their objective strengths are all
reduced.
Yet there is more to monitoring these resources than meets the eye, as there is no single metric which can accurately reflect the state of any one of them. This is particularly true for memory allocation, wherein an attack may monopolize a small but vital memory structure without having a perceptible impact on any system-wide measurements. Different attacks will target different memory structures in different ways, meaning we may need to test dozens of metrics to find one that signals a given attack.

Furthermore, it is important to account for the resources required by the measurement utilities themselves, and to adjust the precision with which data is collected accordingly. Measuring more often does not necessarily provide more accurate results. For example, polling the server’s CPU utilization costs some number of CPU cycles. Doing so too frequently effectively constitutes the server launching a DoS attack on itself, driving up its CPU utilization in a way that is not indicative of the threat posed by the attack being tested. It took a great deal of trial and error to determine the appropriate granularity with which to track each of our metrics. Researches who utilize our software on different hardware, track different metrics, or test different attacks should experiment to find appropriate values for these variables. In our experience, it is generally better to measure infrequently and average results over a large number of trials in order to preserve accuracy.

We use the nmon utility (version 16g) to monitor CPU utilization, network activity, and course-grained (system-wide) memory utilization.[11] More fine-grained memory data – most notably the number of allocated and active objects of each size – is measured by polling the contents of /proc/slabinfo, which we append to a file once per second. Even finer-grained, process-specific memory instrumentation can be performed with ftrace⁴, and process-specific CPU utilization is measured with the perf utility.

⁴While we have implemented support for taking measurements with ftrace, we do not present results from it in this thesis. It generates incredibly large amounts of data, which uses considerable server resources and is challenging to parse. Our preliminary experiments with it yielded no useful information, but we expect that with proper tuning it will eventually prove to be helpful.
3.4 Selected Attack Types

Ultimately, we would like to see all manner of attacks tested with our software, and have explicitly designed it to make adding new packet types and even new protocols as simple as possible. As a proof of concept, we conducted experiments to measure the impact of transport-layer packet length on the effectiveness of DoS attacks.

Perhaps the only characteristic shared by all network packet types is that they have some finite length. The specification of any network protocol must therefore prescribe an allowable length or range of lengths for its packets. Attackers are constrained by both the number of packets and the amount of data that they can send in a given time interval. A given attack type may be most effective when comprised of many small packets or a few large ones, depending on how each packet is handled by the targeted server. We aim to facilitate the design of DoS-resilient protocols by providing guidance on determining optimal packet length limitations.

Selecting an appropriate attack packet type to isolate the impact of packet length is challenging. We cannot expect to draw any valid conclusions by making comparisons across disparate protocols (e.g. comparing the impact of a SYN flood with 20-byte packets to that of an ICMP flood with 64-byte packets) as differences in results may stem from other factors (e.g. different system resources may be required to handle SYN and ICMP packets). Thus we need to use protocols which permit multiple lengths for a single packet type. We must also use packet types that will not be immediately discarded by the recipient, such as a TCP data packet that does not correspond to an existing association, or an IP packet with a random payload that does not conform to any transport-layer protocol’s specification. Options which meet both these criteria are limited, but we have identified and implemented two so far.

UDP Flood

The User Datagram Protocol (UDP) is extraordinarily simple – it enables the best-effort transmission of a message from one host to another, without providing any extra features like mobility or reliable ordered delivery.[77] UDP has a fixed-size header with
four fields: Source Port, Destination Port, Length, and Checksum. This is followed by an optional payload of up to 65,507 bytes\(^5\) of data.

In a UDP flooding attack, each packet is addressed to a different destination port on the targeted server. For each packet it receives, the server must check whether it has an application listening on the specified port, and replies with a 28-byte ICMP Destination Unreachable packet if not. Again, this consumes both incoming and outgoing network bandwidth, and requires CPU utilization in order to check whether applications are running and to compose response packets. We test attacks with payloads containing 0, 8, 64, 512, and 2048 bytes of random data.

**INIT Flood**

This attack leverages the Stream Control Transmission protocol (SCTP), a relatively new transport-layer protocol originally designed to send Public Switched Telephone Network (PSTN) signaling messages over IP networks.\(^{95}\) Despite this narrow vision, SCTP is very versatile and feature-rich. We have been unable to find any verifiable data on the scale of its deployment. Anecdotal evidence suggests it is used almost exclusively within cellular networks, but it was also recently adopted for use in the data layer of WebRTC which we expect will increase its prevalence on the Internet dramatically over the coming years.\(^{53}\)\(^{99}\)

The most interesting feature of SCTP for our purposes is that its initialization procedure was designed specifically to combat a common attack against the Transmission Control Protocol (TCP) \(^{82}\): the SYN flood.\(^{42}\)\(^{43}\) TCP associations are initialized with the following three-way handshake:

- The client sends a **SYN** packet to request the establishment of an association.

- The server sends a **SYN-ACK** packet acknowledging that it has received the **SYN** and would like to proceed with the association.

- The client sends an **ACK** packet acknowledging that it has received the **SYN-ACK** and is ready to begin transmitting data.

\(^{52}\)\(^{16}\) minus 20 bytes for the IPv4 header and 8 bytes for the UDP header.
The flaw in this protocol is that as soon as a server receives a SYN, it allocates some memory for the association. An attacker with access to a botnet or with the ability to spoof the source address of her packets can send a flood of SYNs from different addresses without waiting for any SYN-ACK replies, forcing the server to allocate space for each one and thereby exhausting the memory it needs to establish connections with legitimate clients.

SCTP circumvents this attack by delaying the allocation of state until the client’s IP address has been confirmed, using the following four-way handshake (a comparison between the two handshake protocols is illustrated in figure 3-3):

- The client sends an INIT chunk\(^6\) to request the establishment of an association.
- The server creates and cryptographically signs a “state cookie” containing all the information it needs from the INIT. After sending this to the client in an INIT-ACK chunk, it deletes all state related to the association.
- The client sends the state cookie back to the server, unmodified, in a COOKIE-ECHO chunk.
- The server verifies that the state cookie’s signature is its own, and that the source address of the COOKIE-ECHO matches the one it copied from the INIT into the state cookie. If so, it responds with a COOKIE-ECHO chunk to confirm that it is ready to begin transmitting data.

Delaying the allocation of state is not a perfect solution however, like most mitigation strategies it is a tradeoff. Generating and signing the state cookie requires the server to perform a certain amount of computation. Thus, if a server has ample memory but limited computing power, then SCTP’s four-way handshake design may actually prove detrimental to its security. Moreover, the server is still required to allocate some memory temporarily while it constructs the state cookie, and to use some bandwidth for sending the INIT-ACK. In fact, interplay with the protocol’s mobility features enables the attacker to waste additional resources as well.

---

\(^6\)SCTP packets are modular, each one consisting of one or more “chunks” of various types.
Mobility in SCTP works as follows:

- The client has an option to include in its INIT one or more Address Parameters, each containing a single IPv4 or IPv6 address at which it claims to be reachable.

- Similarly, the server may include one or more Address Parameters in its INIT-ACK.

- If either party detects that the other’s primary address has become unreachable or slow to respond during the course of their association, it may begin sending traffic to one of the secondary addresses instead. Or, one party may preemptively request that the other begin using one of its secondary addresses, if for example it is running on a mobile phone and preparing to switch from Wi-Fi to a cellular network.

- Hosts send periodic heartbeat messages to their correspondents’ secondary addresses, in order to ensure that they are still active and to check if they may be faster than the primary address.

The number of Address Parameters an INIT or INIT-ACK packet may contain is limited only by the Maximum Transmission Unit (MTU) of the networks connecting.
the two parties. We leverage this feature to vary the length of attack packets in our experiments, with the client including varying numbers (0, 30, 60, 90, and 120) of randomly generated IPv4 addresses in its INIT packets. We refer to the packets containing these Address Parameters as Long-INITs, and to the attack leveraging them as a Long-INIT flood or Long-INIT attack.

The specification for SCTP includes an important ambiguity around the handling of these parameters – it states that the server must send heartbeat messages to them to verify their authenticity and availability, but does not indicate the proper time to do so. It seems intuitive that this should not be done until after the INIT’s source address has been confirmed, per the intended purpose of SCTP’s 4-way handshake, yet the protocol’s current Linux implementation opts to begin sending heartbeats to secondary addresses at the same time the INIT-ACK is sent to the source address (before the COOKIE-ECHO is received). This allows the attacker to waste even more of the server’s outgoing bandwidth by including as many Address Parameters as possible. Worse yet, each of these heartbeats must contain a unique random nonce, which requires a non-negligible amount of computation to generate. Figure 3-4 illustrates this attack, and Section 4.4 provides details on the steps we took to identify its exact mechanism.

3.5 Mitigations

The best way to mitigate the effects of a denial of service attack is to prevent it from happening in the first place. Our hope is that this research will facilitate a future in which network protocols and architectures are designed to be intrinsically resilient to attack, such that even the most well-resourced adversary cannot cause a significant service disruption. This goal remains a long way off though, and it is important not to discount other current approaches. A great deal of effort is put into prevention in the form of systems security – by preventing devices from being taken over by hackers, we can reduce the size of botnets and make it impossible to carry out large-scale distributed attacks. This is a perpetual arms race though, and it seems...
Figure 3-4: The Long-INIT attack, in which a malicious client sends a flood of INIT packets with large numbers of secondary addresses, forcing the server to send a heartbeat with a unique random nonce to each address listed. Elements which relate to the waste of server resources are shown in red.

inevitable that a significant portion of devices will continue to remain vulnerable to compromise.

Assuming malicious actors will always find a way to launch attacks, the next best option is to prevent their traffic from reaching its destination. Attacks that rely on address spoofing can be defeated by deploying ingress filtering – if an attacker tries to send packets with an address they don’t own, their Internet Service Provider (ISP) will simply drop the traffic before it reaches its target. Internet transit providers may use sophisticated methods to protect their customers from receiving malicious traffic as well. They may even go as far as using deep packet inspection and machine learning techniques to examine and classify packets in the network.[89]

Yet this intercepted traffic still causes harm to the Internet – even if it doesn’t reach its destination it still wastes some resources of devices along the way.⁷ Thus,

---

⁷The value of drawing a distinction between endpoint and network devices in this context is debatable. In our view, the two are identical in theory but have historically differed in practice. All Internet-connected devices have the same basic resource types, but switches and routers don’t generally run the same set of protocols that servers, desktops, and mobile devices do. Protocols running in the network tend to be simpler and less stateful. This is by design, following the end-to-
we tailor our mitigations to determine how much damage an attack can cause to a system even if that system is able to detect precisely which packets are and are not malicious. We set the Source Port field in our attack traffic to 666, a value we do not expect to see from any other traffic in our test environment. Our server can then utilize a simple iptables rule to drop all incoming packets from that port to effectively block the attack. As we will show, even this “perfect” mitigation can’t negate an attack entirely – the server must still expend some resources just to receive packets and determine whether they match this rule.

3.6 Generating Attack Traffic

There are many factors to consider in determining how best to generate the packets that comprise our attacks. Our primary objectives in weighing these factors are to model real-world attacks as realistically as possible, and to enable a one-to-one comparison between different attack types by ensuring that they reach the server at the same rate. Due to resource constraints, we would also like to be able to launch the largest attack possible using only a small number of hosts.

3.6.1 Address Spoofing

Most major DoS attacks in the modern era are massively distributed, with a large number of devices each sending a small stream of packets. We aim to approximate this threat model as closely as possible, but lack the resources to deploy a large number of attackers. Instead, we use just 25 attacker hosts sending at a fairly high rate and have them spoof the source of their packets with random IPv4 addresses. This makes it appear to the server as if the attack is being launched by a large number of low-rate, widely distributed hosts. It should be noted that real-world attacks which

---

end principle presented in the seminal work of Saltzer et al.[90] This means that the most viable attacks against the network are likely to be bandwidth related, while it may be easier to exhaust an endpoint’s CPU or memory (simply because endpoints run a wider variety of code which may allocate those resources). We expect our testbed to be most useful in securing endpoint devices, but see no inherent reason it could not be extended to test network devices as well.
rely on address spoofing are becoming less and less viable as a growing percentage of networks employ ingress filtering.[25][44]

Even in our test environment, spoofing is non-trivial. By default, OpenStack performs a form of address verification on packets which is akin to ingress filtering. If a host attempts to send a packet from an address that has not been assigned to it by OpenStack, it will be dropped before it reaches its destination. Fortunately this feature can be disabled, but doing so is not entirely straightforward. We omit the technical details here, but a full walkthrough of the process is provided in the documentation for our testbed software.[37]

The second issue is that the Ubuntu operating system does not provide an easy way to send spoofed packets. This is not a shortcoming, but rather a security feature. While it has a very legitimate use case for us as security researchers, address spoofing also has massive potential for abuse. It can enable malicious actors to conceal their identities, and makes their attacks harder to block. We are able to implement address spoofing by using python to manually construct packets and send them over a raw socket. This lets us specify the exact contents of every field in each packet, including the IP header. When not sending over a raw socket, the operating system will construct its own IP header containing its actual IP address.

Generating packets in this way requires a deep familiarity with the protocols being used. For a packet to be handled properly by the server, it must conform to the protocol specification exactly. This is not a large burden for simple protocols like UDP whose packets contain only a handful of fields which carry very clearly defined values, but conforming to the specification of more complex protocols can be quite a challenge. The core specification alone for SCTP is 152 pages long [95], and doesn’t include details on dynamic address reconfiguration for mobility support which comprise an additional 41-page document [65]. For comparison, the UDP specification is just 3 pages long.[77]
3.6.2 Checksums

We found the most challenging aspect of programming both attack types we tested to be correctly implementing the checksum function. Checksums are used to ensure that packets are not corrupted in transit, in approximately the following manner:

1. The sender generates a packet they would like to send, with the Checksum field initialized to zero.

2. The sender computes some sort of one-way hashing function on the binary representation of the packet, and places the result in the Checksum field.

3. Upon delivery of a packet, the receiver copies down the value in the Checksum field.

4. The receiver then sets the Checksum field to zero and computes the same hash function that the sender used.

5. If this computed value does not match the saved value, then the receiver assumes the packet was corrupted or otherwise modified along the way and drops it.

Different protocols use different hashing functions for their checksums, and these functions can be very difficult to communicate clearly in the text of a protocol specification. We discovered a python library called pyip [54] which provides checksum functions\(^8\) for several popular transport protocols\(^9\), but found that they are not always reliable when used in conjunction with address spoofing. This is because some checksum computations take input from the IP “pseudo header” including the packet’s source address, and pyip does not allow us to insert our fraudulent address in advance. Swapping the address afterwards only serves to invalidate the checksum. Fortunately, it is easy to verify that our checksum implementations are correct: we send a test packet from an attacker to the server and use tcpdump to monitor the server’s incoming network interface. If it receives a packet containing a bad checksum, tcpdump will

---

\(^8\)This is just one of many features – it defines a python class for each protocol type, subclasses for specific packet types which pre-fill default values, and functions for basic operations like converting a packet between its structured and binary representations.

\(^9\)It does not provide support for SCTP.
print a message attesting to that fact. This feature is even implemented for SCTP, despite its limited use.

### 3.6.3 Rate Limiting

The rate at which networked devices are capable of sending and receiving packets is limited in term of both bytes- and packets-per-second. The bytes-per-second limitation is much more obvious and well-known. Network cables have only a finite capacity, so the theoretical maximum rate at which information may be sent from one host to another is limited by the capacity of the most narrow link along the path between them. ISPs typically place artificial byte-per-second\(^{10}\) restrictions on their customers as well. This prevents a small set of overactive devices from consuming a disproportionate percentage of their network’s capacity, and enables the ISP to charge a higher price for a faster connection.

The packet-per-second limitation is much less cut and dry. It is an artifact of the kernel’s network stack implementation – the code that Linux (in our case) must run to process each incoming and outgoing packet takes a non-zero amount of time to execute, and typically can only process a single packet at a time. A skilled application designer may be able to increase their effective packet handling rate by enabling parallel packet processing for systems with multiple CPUs, but doing so takes a considerable amount of effort and is not necessarily viable for all application types.\(^{63}\) Our experiments are all conducted on single-core machines, so even if we could implement these optimizations they would have no effect on our results. We found that the Linux kernel version we tested was capable of sending or receiving at most \(\sim 150,000\) packets per second.

The packet-receiving rate of our server is a legitimate but often overlooked resource constraint; one which may play a pivotal role in shaping the differences between disparate attack types. Bumping up against the maximum packet-sending rate on our attackers could be problematic though. Recall that we are effectively simulating a large number of low-rate attackers with a small number of high-rate ones. Actual

\(^{10}\)Usually advertised in terms of megabytes per second (MB/s)
low-rate attackers would almost certainly fall well short of their ∼150,000 packet-per-second limit, for risk of detection.\footnote{If your smart toaster stops working because it’s using all its resources to help launch a DoS attack, you are likely to discard it and replace it with an uninfected device, even if you’re unaware of the precise issue preventing it from working.} We would therefore expect the rate at which packets arrive at the server to be bounded only by the number of attackers and the capacity of the intervening network links. Thus, even though a server’s byte-per-second and packet-per-second limits are both important, drawing comparisons between attacks requires us to deliver the same number of bytes-per-second to the server with each one.

With a ∼150,000 packet-per-second limit, each attacker can send ∼78 MB/s of our longest (520-byte) packets, but only ∼1.2 MB/s of our shortest (8-byte) packets. This forces us to limit all attacks to ∼1.2 MB/s per attacker or less, which means we need a large number of attacker hosts to generate a sizeable flood. It’s worth noting that even with multiple hosts sending attack traffic, our floods are very small in comparison with real-world DoS attacks at a total volume of just ∼30 MB/s. They are not able to disrupt the client’s connection entirely, but even a small drop in performance allows us to observe how our different attack types compare.

It is important to implement rate limiting properly in order to produce a large and consistently sized flood. Initially we set limits within the python script used to generate our packets, by recording the time at which each packet was sent and waiting the appropriate interval before sending the next one. In retrospect, the flaw in this approach is quite obvious: checking the current time is a simple operation, but checking it ∼150,000 times per second is extremely costly. Our attack slowed to a crawl as it spent more time computing how long it needed to wait than it should have spent actually waiting. Ultimately, we turned to the Linux traffic control (tc) utilities to rate-limit all traffic on our attackers’ outgoing interfaces. The tc command provides several different types of rate limiters, each with a high degree of customization, but we found the simplest one with the default options to be sufficient for our purposes. We implement a rate limit of 10Mb/s using just the following one-line command:
sudo tc qdisc replace dev ens3 root netem rate 10mbit

Here 10mbit is the rate we set, and ens3 is the name of the network interface we’re limiting. These two values are the only components of this command that users of our testbed may need to modify, everything else can be copied verbatim (assuming the testbed is running on Linux systems).

3.6.4 Randomness On-Demand

Every packet we generate has at least one randomized component, and it is important for this to change from packet to packet for the attack to have its full effect. These components include the spoofed source IP addresses in all attacks, the payload and destination port numbers in UDP floods, and secondary addresses in Long-INIT floods. System resources are often allocated on a per-address or per-port basis, so randomizing these values ensures that our attack consumes the maximum amount of its victim’s resources possible.

Generating random values requires a small but non-zero amount of time, so generating them on the fly while we launch our attacks will reduce our maximum attack rate. Moreover, different attack variants require different amounts of randomness for each packet – an INIT packet with five secondary addresses requires less time to generate than one with a hundred addresses. Though small, this difference can compound over the duration of an attack to have a significant influence on the results.

This raises an important question: should we try to eliminate this difference to ensure that different attack types all reach their target at the same rate, or is it an important factor in comparing attacks that we should retain and measure? We do not have a definitive answer to this question, and expect that it may depend on the particular threat model in question. In the case of a truly distributed DoS, attackers will not need to spoof IP addresses. Each attacker could also be configured to use a different, fixed value in other fields to maintain randomness across the flood. Thus our assumption is that eliminating the effect of randomness generation from our results will provide a more accurate reflection of most real-world attacks. In an effort to be
as accommodating as possible, our test software provides both options. Users may choose to generate each packet on-demand as the attack runs, or to pre-generate a list of packets (with a configurable size to determine the degree of randomness) and cycle through them one by one. We opt to use the former option in our experiments as it seems to provide more consistent results – quickly reading packets from a file can cause performance spikes for the attacker.
Chapter 4

Experimental Results

As a proof-of-concept for the testbed environment designed in Chapter 3, we have conducted an initial set of experiments which test variants of the two DoS attack types described in Section 3.4: UDP and INIT floods.

We selected these particular attacks because they both support variable-length packets. Perhaps the only attribute that all network protocols share in common is that their packets have some finite length, and to our knowledge no prior work has conducted an empirical examination of the relationship between the length of packets in a flooding attack and the potency of that attack. Additionally, these attacks both utilize the first packet type that is sent in the process of setting up an association. This is a requirement for most, if not all viable attacks that employ source address spoofing. Address spoofing prevents an attacker from being able to receive a reply to their packets, which in turn prevents them from establishing an association. Attacks which utilize non-initial packets generally require an ongoing association, as otherwise the server will not be expecting such packets and can simply drop them without having to expend a significant amount of resources.

We first provide instructions for interpreting the figures which present our results. We then show the impact of attacks on the quality of service for a connection between the client and server (as determined by metrics described in Section 3.3.1), followed by a coarse-grained examination of the server’s resource utilization (using metrics described in section 3.3.2). Finally, we present plots of finer-grained resource
utilization and explain how they can be used to diagnose the root causes underlying DoS vulnerabilities. These enable us to pinpoint the exact line of kernel code responsible for SCTP’s vulnerability to Long-INIT attacks: a premature call to generate random bytes, likely implemented because of ambiguous language in the protocol’s specification.

4.1 Interpreting Plots

All plots presented in Sections 4.2 and 4.3 below follow the same general structure. Each is divided into two subplots by attack type, with UDP on the left and INIT on the right. Time is plotted on the x-axis and the metric in question on the y-axis. The red vertical line shows the time at which attackers begin sending malicious traffic. The green vertical line represents the time at which the mitigation against that attack is implemented.¹ For throughput plots, the target rate of the client-server connection is indicated by a dashed black line. Each variant of an attack type is plotted as its own line in a different color, as indicated in the legend. The number associated with an attack variant corresponds to payload size in bytes for UDP attacks, and to the number of IPv4 Address Parameters for INIT attacks. For each attack variant, the dark line shows the mean y-value while the lighter band around it (in the same color) indicates the 95th percentile confidence interval.

Section 4.4 includes similar line graphs displaying fine-grained memory allocation, as well as several flame graphs showing fine-grained CPU utilization. Flame graphs offer a helpful, interactive visualization of code paths. They can be generated from stack traces captured with the Linux perf utility using tools developed by Brendan Gregg.¹ The y-axis of the graph shows stack depth, with call hierarchy ordered from bottom to top. In other words, the function beneath a given function is its parent and all functions above a given function are its children. The x-axis does not

¹Due to the lack of clock synchronization among hosts, these markers may be a few seconds off from the actual start times of attacks and mitigations. Implementing clock synchronization would require sending a considerable amount of extra traffic over our network which would likely interfere with our experiments.
represent time, and in fact position along it has no real meaning; processes are simply sorted alphabetically from left to right. Color also has no meaning; the different hues are picked at random from a given palette to help the eye differentiate between adjacent boxes. The width of each box can represent different things depending on what is being profiled. In the case of our CPU traces of an SCTP server, the width of a box represents the frequency with which its function was present in the stack traces.

Flame graphs are typically generated as Scalable Vector Graphics (SVG) files, which can be interacted with in a web browser or other suitable application. Hovering over a function box in the graph will display the function’s full name, the number of times it appears in the trace, and the percentage of the trace it occupies. Clicking on a function box will zoom in along the horizontal direction so that it and its parent boxes span the entire x-axis. This allows both its child and parent functions to be viewed more easily. As we cannot maintain this interactive quality of flame graphs in print, we instead include images of the relevant portions of our graphs, and have made the full SVG files available online at [39] for closer examination. High resolution versions of the other figures in this thesis can be found there as well.

4.2 Quality of Service

The following plots show quality of service measurements taken on the client. Figure 4-1 shows throughput of traffic sent from the server to the client after the client initiates a connection, as measured by iPerf. Figure 4-2 shows jitter for data received from the server, Figure 4-3 shows the percent of packets lost in transmission from the server, and Figure 4-4 shows the round trip time (RTT) from client to server and back. RTT is measured with ping, while the other metrics are all measured with iperf. Note that these two utilities must be used in separate runs in order to prevent them from interfering with one another.

Under the INIT attack, longer attack packets reduce client performance more severely for all metrics except RTT. For UDP, shorter payloads provide worse perfor-
mance across all metrics except jitter, upon which packet length does not appear to have any impact. These results clearly show that packet length does have an impact on the potency of DoS attacks, and that the nature of that impact is protocol-specific. An attacker leveraging UDP would be better off sending a large number of short packets, while more damage can be dealt to an SCTP server by using fewer longer INIT packets.\footnote{Unless of course the application utilizing SCTP prioritizes latency above all else.}

It’s also worth noting that our simplistic mitigation strategy of dropping malicious traffic is not very useful in defending our server, particularly against UDP floods. The server is still forced to consume a damaging amount of resources while this mitigation is deployed. This proves that the ability to detect and filter out malicious packets is not necessarily an effective defense against DoS attacks – considerable damage may already be done before a packet can be dropped. More details on our mitigation method and the reasoning behind it can be found in Section 3.5.

![Figure 4-1: Throughput from server to client.](image)

### 4.3 Course-Grained Resource Utilization

Now that we’ve seen which attack types are most damaging to the client’s quality of service, we analyze the server’s bandwidth, CPU, and memory utilization to diagnose the mechanism underlying each vulnerability. Note that the data presented here is all
Figure 4-2: Jitter of connection between server and client. We have not yet been able to determine the cause of the periodic spikes seen in these plots.

Figure 4-3: Percentage of packets sent by the server that are not received by the client.

recorded during runs with the iperf client, but data from the ping client looks nearly identical since attack traffic (not client traffic) is responsible for the vast majority of resource utilization.

Figures 4-5 and 4-6 show network read operations in KB/s and packets per second, respectively. The current Linux network stack cannot read more than \( \sim 150,000 \) packets per second, so shorter packets cause the server to read less data per second even though the same amount is delivered by attackers in all cases. For UDP though, we can see that the degradation of client performance is driven by packet reads rather than the number of bits read per second. For SCTP, an increase in bits read per second is correlated with worse client performance, but not very strongly.
Figure 4-4: Round Trip Time from client to server and back.

Figure 4-7 shows the percentage of system-wide CPU utilization. For all attack types, shorter packets consume more CPU. Figure 4-8 shows system-wide memory activity. As expected, the UDP attack has no visible impact on the server’s memory. The SCTP attack clearly does, and that impact is roughly correlated with packet length.

Figure 4-5: Kilobytes read per second by server network interface.

4.4 Fine-Grained Resource Utilization

Our first approach to diagnosing the mechanism behind SCTP’s vulnerability to INIT floods was to monitor the /proc/slabinfo file. Those results, presented in Figures 4-9 through 4-16 ultimately proved inconclusive. We can see that certain-sized memory
objects are used more heavily than others during the attack, but there is no apparent correlation with packet length.
Figure 4-9: Number of active 192-byte memory objects.

Figure 4-10: Total number of 192-byte memory objects.

Figure 4-11: Number of active 256-byte memory objects.
Figure 4-12: Total number of 256-byte memory objects.

Figure 4-13: Number of active 512-byte memory objects.

Figure 4-14: Total number of 512-byte memory objects.
Figure 4-15: Number of active 1024-byte memory objects.

Figure 4-16: Total number of 1024-byte memory objects.
Next, we used the Linux `perf` utility to record stack traces showing kernel process-level CPU utilization, and generate flame graphs to illustrate which processes are responsible for SCTP’s vulnerability. Since there is no way to present data from multiple attacks in a single flame graph, we compare only our longest and shortest INIT attacks here.

Reading the two graphs in Figure 4-17 from the bottom up, we see a difference in the split between `ksoftirqd/0` (which we have highlighted with a black box to increase visibility) and `swapper`. While these two functions eventually converge to call the same set of functions (starting with `_softirqentry_text_start`), we can see that longer INITs cause a larger percentage of calls to start from `ksoftirqd/0`, a process that runs when under heavy soft-interrupt load.

Looking farther up the stack (on both the `ksoftirqd/0` and `swapper` sides, we find another notable difference. Figure 4-18 shows portions of the graphs in Figure 4-17, zoomed in to highlight the functions called by `sctp_rcv`. Here we can see that the `sctp_process_init` function (which we have highlighted with a black box to increase visibility) is using considerably more CPU time while under attack by a flood consisting of longer INIT packets. This single function accounts for 8.11% of the systems CPU usage when under attack by short INIT packets, and 45.27% when under attack by long INITs. Note that these numbers do not indicate a percentage of the systems total CPU resources, but rather how the resources that are in use are distributed between processes.

This gives us an idea of where in the kernel code we should start looking to pinpoint the root of the problem. The definition of the offending function (`sctp_process_init`) begins on line 2305 of the file `/net/sctp/sm_make_chunk.c` in our Linux kernel version (4.15.0-22-generic).[7] For each address parameter in an INIT packet, this function calls: `sctp_process_param → sctp_assoc_add_peer → sctp_transport_new → sctp_transport_init`. Here we finally find our culprit: on line 97 of the file `/net/sctp/transport.c`, Linux runs the `get_random_bytes` function to generate a 64-byte nonce.[8] Recall that this is being done for every single address parameter in each INIT packet the server receives, before it has had a chance to verify the source
Figure 4-17: Server CPU utilization while under attack by INIT packets with no address parameters (top) and INIT packets with 130 IPv4 Address parameters (bottom).
Figure 4-18: Zoomed in views of the two graphs in figure 4-17, showing the breakdown of CPU usage for processes called by sctp_rcv. Again, the top plot shows usage while under attack by INIT packets with no address parameters, and the bottom shows usage while under attack by INIT packets with 130 IPv4 Address parameters.
address of that packet. This premature computation, which scales with the number of address parameters, is the precise reason why SCTP is more vulnerable to INIT floods comprised of longer packets.

This is a flaw in the Linux implementation of SCTP, not in the protocol’s design. However, the implementation does still conform to the specification – the problem is that the specification contains ambiguous language which makes it unclear exactly when these random nonces should be generated. It states:

“To probe an address for verification, an endpoint will send HEARTBEATs including a 64-bit random nonce and a path indicator (to identify the address that the HEARTBEAT is sent to) within the HEARTBEAT parameter.”

It indicates that this probing should be done for any “UNCONFIRMED” addresses, but does not state exactly when it should begin. Later, it is stated that:

“HEARTBEAT sending MAY begin upon reaching the ESTABLISHED state and is discontinued after sending either SHUTDOWN or SHUTDOWN-ACK.”

The word “MAY” here (in all-caps) is defined by the IETF as referring to something “truly optional”.[29] The intended meaning was likely that Heartbeats MUST NOT³ be sent prior to entering the ESTABLISHED state, but as written it could be interpreted to mean that they may be sent earlier.

This vulnerability perfectly exemplifies the complexity of protocol security. The design is sound, but one minute piece of ambiguity in the specification was able to produce an implementation that is simultaneously correct and flawed. Ironically, the random 64-byte nonce was actually intended to improve the security of SCTP. It was introduced when the specification was updated from [97] to [95] in order to defend

³Also as defined by [29].
against address camping, another type of Denial of Service attack.\textsuperscript{4} We were only able to uncover the negative side effects of this well-intentioned modification to the protocol’s design by watching multiple different attacks in action and taking detailed measurements of the targeted server’s resource utilization.

\textsuperscript{4}Interestingly, Rathgeb et al. actually conducted experiments very similar to ours on the effect of Long-INIT flooding attacks in [84], but used an older Linux kernel version which had not yet incorporated this update into its implementation of SCTP. This led them to conclude that attacks using INITs with secondary address parameters were less damaging than those without. These contrasting results illustrate the importance of testing on up-to-date systems, and of periodically reproducing existing research. In fact, the original impetus for this thesis was our failed attempt to reproduce the results in [84].
Chapter 5

Discussion

Having seen the results of our experiments on packet length, we next discuss their potential implications for the domain of protocol design. We then give an overview of how our software works, why we made certain design decisions in both our code and data storage, and a set of variables that it is currently capable of evaluating. Finally we touch on the responsibility that comes along with building and using software such as ours, which has the potential to cause serious harm if misused.

5.1 Packet Length Guidelines

Based on the results presented in Chapter 4, we propose a categorization of protocols based on how the resources required to handle an unsolicited packet scale with that packet’s length. Protocols like SCTP, which must expend more effort to process larger packets, should be treated differently from those like UDP which have a fixed cost for handling any initial packet, no matter how long it may be. Thinking about protocols in this way offers a new perspective on designing in DoS resilience. When possible, protocols should be designed to avoid working harder to handle longer packets, particularly ones from untrusted sources. If a required feature of the protocol makes this impossible, designers should strongly consider imposing strict limits on packet length, especially for packet types that are used to initiate a new association with a potentially unknown party.
Specifically for SCTP, we recommend three possible ways to meet these guidelines and reduce its vulnerability to attack:

1. Clarifying in the specification that the server MUST NOT begin sending Heartbeat packets to secondary addresses until after it receives the COOKIE ECHO and is able to verify its peer’s source address.

2. Having clients indicate only the quantity of secondary addresses they would like to use in the INIT, and waiting until the COOKIE ECHO to list the actual addresses (after the client’s source address has been verified).

3. Placing an upper bound on the number of address parameters allowed in an INIT packet. This would not meaningfully alter functionality as additional addresses can be added to an association after it has been initialized.

A similar modification for UDP (enforcing a higher minimum packet size) is not advisable. Doing so would require wasted bandwidth when sending short messages and more frequent re-transmissions on lossy or congested links. Furthermore, our naive mitigation against short UDP packets had no significant effect – even if the specification required a minimum length, attackers could likely still exhaust the server’s resources by sending packets that do not comply with the specification.

5.2 Testbed Software

One major contribution of this work is the software we have developed for our testbed environment, which is freely available at [37]. Detailed instructions for its installation and operation are annexed to the documentation that is included with and in our code, but there are aspects of the software’s design worth describing here.

5.2.1 Code Structure

Our code is a combination of Bash [83] and Python [14]. Each host has an identical set of files. This structure means that some hosts possess code they will never use,
but removes the hassle of remembering to update individual hosts as code is added or changed – they all simply pull from a single git repository. As such, we strongly advise against storing experimental results (or any other large files) in that same git repository. Doing so would make updating hosts an increasingly slow process as the database grows. Instead, the bridge gathers data files from each of the other hosts once it finishes a run. Details of our data processing and storage procedures are discussed in section 5.2.2.

Users may set all configurable parameters using a single bash file, config.sh. Once parameters are selected and pushed to each host, a set of tests is initiated from the Bridge by running start.sh X where X is the number of test iterations (“runs”) to perform. This script calls and logs the output of a larger bash file called run.sh, which executes the experiments.

Attack traffic is all generated by flood.py. Each attack type is defined as a bash script in the “attacks” folder, which effectively serves as a preset – calling flood.py with an attack-specific set of command-line arguments. Client and Server applications are defined by bash scripts in the “clients” and “servers” directories, respectively. Mitigation strategies are defined by bash scripts in the “mitigations” directory. The particular attack, client, server, and mitigation types to use for a set of experiments are specified in config.sh. Once experiments have finished, users may run the python program plot.py to parse the data files, add useful information to a database, and automatically generate a set of graphs.

5.2.2 Data Structure

Each row in our database represents a single data point, and each column corresponds to a different variable: an x-value for the time, a y-value for a particular metric’s reading at that time, the name of that metric, the type of attack used, the rate of the attack, etc..

This allows us to combine data from many different runs with varying parameters into a single database, expanding it over time as we test more and more configurations. When plotting figures, we can then pre-filter the database to only include
(for example): data from a single configuration, data from all configurations with a certain attack rate, data from a particular set of attack types, etc.. We can also compare across different values of any of these variables as we have done with attack type and packet length in Chapter 4.

Our hope is that this data structure will facilitate collaboration – if many different researchers use our software to collectively construct a single large dataset, they can include variables identifying the group responsible for a give set of experiments, or the specifications of the hardware and kernel version they use. This would allow datapoints from sufficiently similar runs to be averaged together to improve accuracy, while avoiding merging results from drastically different configurations.

5.2.3 Testable Variables

The experiments we have conducted so far focused primarily on the effect of attack type, and specifically packet length. Yet our software is designed so that users can test a wide range of variables, and provides the following capabilities via its configuration file:

- Selecting the type and contents of attack packets
- Selecting a client/server application
- Selecting a set of resource monitoring utilities
- Setting the granularity of measurements
- Selecting a mitigation strategy
- Setting the timeline of a given run (duration of client connection, delay before attack, duration of attack, time to wait before deploying mitigation, etc.)
- Setting the congestion control protocol used for the client-server connection
- Setting the IP version (IPv4 or IPv6) of attack and client-server traffic
- Setting network rate limits for each host type
• Setting the number of attackers
• Enabling/disabling source address spoofing and randomization for attackers
• Enabling/disabling DNS lookups for each host type
• Setting the MTU for the network interface of each host type

Note that certain variables have a more restricted range than others in our current implementation. Naturally, things like attack duration, rate limits, and measurement granularity are all integer or floating point values which can be adjusted arbitrarily. Adding a new server resource utilization metric requires writing a bash script that initiates that metric and pointing `run.sh` to it. Adding a new attack type can vary in complexity. If it uses an already implemented protocol, it may be possible to specify it as a new “preset” file which passes a particular set of command-line arguments in a call to `flood.py`. If not, users will first need to define a new class for their protocol’s packet structure in `flood.py` and add a command-line argument to utilize it.

5.3 Ethical Considerations

It is vitally important to be cautious and ethically responsible when conducting security research of this nature. In the wrong hands, the code we have written is capable of launching potentially devastating attacks against real-world systems. That said, we do not expect our software to be particularly appealing to malicious actors. The UDP flood we implemented is very well known, and other implementations of are not difficult to find. The INIT flood is less well-known, but likely because it effects a protocol that is rarely used by public-facing servers and therefore has limited use to those seeking to take production systems offline. According to our QoS metrics, even our Long-INIT attack is less powerful than the other, simpler attack types. Thus, we expect the value this test environment provides to the scientific community to far outweigh any harm it may be used to inflict. The most novel contributions of our software are not the attack implementations, but rather the framework for automating experiments and analyzing the results.
Well-intentioned users of this software must also take care not to cause accidental damage with it. Experiments should be run on a network that is as isolated as possible. Attack traffic should only ever be sent to one’s own devices and should only be transmitted over one’s own network. When using source address spoofing, one must be wary of how the server will respond to the attack packets it receives. If it sends replies to someone else’s device, that may constitute an inadvertent reflection attack against them. Ideally, the network should be sufficiently isolated to prevent any test traffic from leaving it.
Chapter 6

Conclusion and Future Work

In this thesis we designed, deployed, and tested a controlled network testbed environment designed to facilitate the empirical analysis of Denial of Service attacks. Our hope is that this testbed will spur more comparable and reproducible research into the effects and mechanisms of different attack types, eventually allowing the community to develop a more comprehensive set of best practices for designing DoS-resilient network protocols.

We began by defining the goal of internet security and explaining why availability is a particularly important aspect of that goal. We gave an overview of the current Internet’s structure, existing efforts to improve its design, and the procedures in place for developing new protocols. These procedures lack stringent guidelines for analyzing protocols’ security guarantees, which prompted us to consider potential tools and methods that protocol designers could be encouraged or required to use. After explaining why static analysis is likely an untenable approach in this domain, we settled on empirical measurement as the most practical way to assess DoS vulnerabilities.

Next, we laid out the plans for a testbed designed to perform such measurements in a controlled and easily reproducible environment. We discussed the importance of measuring both quality of service and resource utilization, the most valuable metrics for measuring each, and the tools we selected to track those metrics. The attack types we chose to implement were intended to illustrate the impact that packet length has on the effectiveness of flooding attacks. Our testbed is general purpose and extensible
though; researchers may augment it with metrics and attacks of their choosing in order to test whichever hypotheses they may have.

The experiments we conducted are more than just a proof of concept, and have real implications for protocol design. They showed that the length of packets in a DoS attack is indeed a meaningful factor in determining the impact of that attack. Furthermore, we saw that this effect varies between different attack types – some are more damaging when made up of a large number of very small packets, while others work better with fewer shorter packets. By analyzing kernel process-level CPU utilization, we were able to precisely identify the cause of a novel vulnerability in SCTP. Based on these results, we proposed specific changes to SCTP as well as some more general guidelines for the design of future protocols.

We then discussed some of the design decisions that went into programming our testbed software and gave an overview of the set of variables across which it allows researchers to test. Finally, we considered the ethical concerns relevant to this work, and the responsibility that both designers and users of software like ours have to avoid facilitating or accidentally initiating DoS attacks against real-world systems.

There is a great deal of work left to be done in the field of designing secure, DoS-resistant network protocols. As a next step, it would be valuable to repeat our packet length-based experiments on other network topologies and with additional protocols in order to strengthen our understanding of this variable’s effect and help develop more concrete design principles. Ideal candidates for such studies must support variable length for packets accepted from untrusted sources – options include ICMP[79] and QUIC[52]. Since QUIC is still in the draft phase of the standardization process, there is particular value in testing attacks against it now so that the standard, once published, is as DoS-resilient as possible. It would also be interesting to run tests over IPv6 as that would impose a considerably larger minimum packet size for UDP attacks, potentially offering a new incentive to expedite the migration from IPv4 to IPv6.

We plan to continue developing our testbed software in order to make it more user-friendly and feature-rich, to support the scientific community as well as our own
future work in this field. The features we are most eager to implement include:

- Support for multiple simultaneous client connections
- Allowing different attackers to send different packet types
- A more flexible method for specifying the timeline of an attack
- CPU metrics that accurately support multi-core servers
- Arbitrary protocol layering in attack packets (currently users can only select a transport-layer packet type and a choice of either IPv4 or IPv6).

We firmly believe that the FIA project [73] is the most promising avenue for someday migrating the world to a new and better internet architecture. A key aspect of that project is determining how to compare novel architectures with one another to determine, for example, which one provides the best protection against DoS attacks. While adding support for non-IP based traffic to our testbed would take a fair amount of work, we believe it to be well worth the effort. It’s important to evaluate novel architectures relative to one another and relative to the current Internet, and doing so will require an architecture-independent testbed design.

Despite the technique’s current limitations, we fully support attempts to formally verify network protocols – specifically those which take into account the transitions between design, specification, and implementation phases. Their success would mark a paradigm shift for the protocol development process. That said, we would caution against excessive optimism in this domain, as some network protocols and internet architectures may simply be too complex to ever be properly verified.

We encourage the community to continue performing empirical experiments, even on protocols that have been “proven” secure, in case their proofs turn out to be flawed or lacking consideration of some variable. Empirical analysis is the best tool we have for observing how networked systems behave in the sort of corner cases that only arise in the midst of a clever attack. The more we learn about the vulnerabilities created by different protocols’ features and the interactions between them, the better prepared we will be to design and build a more secure, DoS-resilient internet.
Appendix A

SCTP Control Flow Diagrams

This appendix contains a set of diagrams we created to illustrate the control flow of the Stream Control Transmission Protocol (SCTP) [95], with the following four goals in mind:

- To gain a better understanding of SCTP’s operation.
- To communicate the potential complexity of a full-featured network protocol.
- To search for any ambiguous language in the protocol’s specification which may give rise to faulty or non-interoperable implementations.
- To search for loops or other structures which commonly cause security issues.

We first provide a legend for parsing the rest of the diagrams, followed by the core state machine of the protocol. All subsequent diagrams define discrete functions, which are each called in either the core state machine or in some other function definition. These are provided in alphabetical order by function name. It should be noted that these diagrams do not contain the full detail provided in the specification document; a few sub-functions are included without a corresponding diagram to define them.

Note that some of the included function definitions come from [65], an addendum to the core protocol which defines the procedure for Dynamic Address Reconfiguration (the ability to change the list of addresses in an ongoing association on the fly).
Figure A-1: This legend explains the visual language of the remaining diagrams in this appendix. Protocol states are written in all-caps within rectangular boxes. Actions are written in rounded rectangles. Conditionals are written in diamonds. Functions are written in double-lined rounded rectangles. Function definitions begin with their name in a double-lined circle. An event that requires waiting for some other event is written in a hexagon. Actions which occur concurrently are connected by double lines. A required transition from one object to the next is shown by a solid line with an arrow. A timeout transition is shown by a dotted line, annotated with the name of the corresponding timer. Optional actions/functions/conditions/transitions are shown with dashed lines instead of solid ones.
Figure A-2: This is the core state diagram of the protocol, expanded slightly from the one provided in [95]. It begins in the CLOSED state shown by the solid black box.
Figure A-3: Procedure for calculating the Retransmission Timeout value [RTO]
Compute Association Shared Key

create two key vectors (one from INIT and one from INIT-ACK) by concatenating all sent RANDOM, CHUNKS, and HMAC-ALGO parameters as byte vectors, omitting all padding

two key vectors are identical

Yes
concatenate endpoint pair shared key + either key vector + other key vector

No

two key vectors are numerically equal but differ in length

Yes
concatenate endpoint pair shared key + shorter key vector + longer key vector

No
concatenate endpoint pair shared key + smaller key vector + larger key vector

set as Association Shared Key

Figure A-4: Procedure for computing an Association Shared Key
Figure A-5: Procedure for establishing an Association Shared Key
Generate State Cookie

- create TCB
- set TCB creation time to current time
- set TCB lifespan to `Valid.Cookie.Life` parameter value
- identify minimum information needed to recreate TCB
- generate MAC using required information and secret key (see RFC 2104)
- create state cookie by combining required information with MAC

Figure A-6: Procedure for generating a State Cookie
Figure A-7: Procedure for handling an "out of the blue" [OOTB] (unexpected) packet.
Figure A-8: Procedure for handling an Unexpected Cookie ECHO chunk. The yellow line indicates the most likely action to take for the given condition, based on some ambiguous language in the specification. Note that this diagram contains several sub-function definitions.
Figure A-9: Procedure for processing an Address Configuration Change [ASCONF] chunk.
Figure A-10: Procedure for processing a DATA chunk.
Figure A-11: Procedure for processing an INIT chunk. Ambiguity in the specification makes it unclear when to begin sending Heartbeat Requests to unconfirmed addresses. The earliest point at which this may occur is shown in red, as doing so then is insecure.
Figure A-12: Procedure for processing a selective acknowledgement [SACK] chunk.
Figure A-13: Procedure for processing Type-Length-Value [TLV] information found in an ASCONF chunk.
Figure A-14: Procedure for sending an ASCONF chunk.
Figure A-15: Procedure for sending a DATA chunk.
Figure A-16: Procedure for validating an AUTH chunk.

Figure A-17: Procedure for validating a Cookie ECHO chunk
Figure A-18: Procedure for verifying an ABORT chunk
Bibliography


