Redistributing the Costs of Volumetric Denial-of-Service Mitigation

by

Samuel DeLaughter

B.A., Hampshire College, 2008
S.M., Massachusetts Institute of Technology, 2019

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

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2023

Authored by:	Samuel DeLaughter Department of Electrical Engineering and Computer Science August 29, 2023
Certified by:	Karen Sollins Principal Research Scientist Computer Science and Artificial Intelligence Lab Thesis Supervisor
Accepted by:	Leslie Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Redistributing the Costs of Volumetric Denial-of-Service Mitigation

by

Samuel DeLaughter

Submitted to the Department of Electrical Engineering and Computer Science on August 29, 2023 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Volumetric Denial-of-Service (DoS) attacks pose a severe and exponentially increasing threat to the Internet. Existing mitigations provide valuable stop-gaps but fail to address the root cause, and the overhead they incur is poorly understood. To combat these attacks we present a protocol-agnostic approach to DoS mitigation that moves overhead away from service bottlenecks, towards the network edge and onto attackers themselves. We observe that the vast majority of attacks rely on a small subset of packet types which are individually identical to legitimate packets, but generated far more often by attackers than by regular clients. Making such packets marginally more difficult to generate can significantly reduce flood volumes without harming legitimate clients. We design and implement two novel mitigations in TCP following this approach, to combat the ubiquitous SYN Flood attack. The first is largely a toy example illustrating how simple packet padding can rate-limit bandwidth-constrained attackers, while the second is a more robust approach using miniature proofs-of-work to restrict the common CPU-bound attacker. We also present a rigorous experimental methodology and novel suite of metrics for more accurately evaluating the efficacy and overhead of arbitrary DoS mitigations across changes in attack, client behavior, and network topology. We use this measurement framework to evaluate our proposed mitigations in a controlled network testbed. Both mitigations exhibit negligible overhead, and while their efficacy is subjective they succeed in completely nullifying potentially devastating SYN floods in certain contexts. Beyond our immediate findings in TCP, this work is broadly applicable to the design of DoS-resilient network protocols and internet architectures.

Thesis supervisor: Karen Sollins
Title: Principal Research Scientist
Computer Science and Artificial Intelligence Lab

Acknowledgments

Completing this thesis has been a wonderful opportunity but also a tremendous challenge. I am eternally grateful to all those who have supported me through this journey. My advisor, Karen Sollins, has provided seven years of unparalleled mentorship. She is kind, patient, infinitely generous with her time, and always knows what questions to ask to "turn a problem on its head." My committee members, David Clark and Mohammad Alizadeh, both provided exceptionally thoughtful feedback on my thesis, and somehow made my defense a calm and pleasant experience. Thanks to Arthur Berger, Steve Bauer, Bill Lehr, Philipp Richter, Cecilia Testart, Leilani Gilpin, and all the other members of ANA and IPRI for their invaluable thought partnership over the years. Thank you to my family, for encouraging my interest in computers and the Internet from a very early age, and for always supporting my pursuit of knowledge. To my wonderful friends, for reminding me to listen to music, play games, laugh, and enjoy the process. Finally, thank you to my beloved partner and best friend, Kristína Moss, for being by my side in this and every other adventure. Without her encouragement I would never have considered applying to this program, and without her endless love and support I would never have made it through.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet has become essential to nearly every aspect of modern society, from commerce to entertainment to national security. With this tremendous utility comes an equally great risk. When large portions of the Internet go down the whole world seemingly grinds to a halt. Sometimes this happens by mistake (like a misconfiguration in BGP or DNS [1]) or by acts of nature (like sharks chewing through undersea cables [2]), but more often than not such outages result from deliberate Denial-of-Service (DoS) attacks. DoS is a broad term that refers to countless, highly diverse attack vectors. Generally, the attacker's goal is to disrupt or degrade some ongoing communication over the network, and/or to prevent new connections from being established. Some attacks use precisely crafted packets to exploit vulnerabilities in protocols or their implementations, but an increasing majority are brute-force volumetric attacks. Ordinary packets are generated in such large quantities that receivers are forced to discard or de-prioritize traffic from legitimate users.

In the past decade these volumetric DoS attacks have become ubiquitous, driven by desire for profit, revenge, chaos, and destruction. As we will discuss in Chapter §2 they are increasing exponentially in both frequency and scale, with hundreds or even thousands of unique attacks now observed every day. The largest floods have reached several terabits per second and millions of packets per second. Individual packets in these attacks are typically indistinguishable from those sent by legitimate users, but each one consumes a small amount of resources at its destination and at each hop through the intervening network. Such flooding attacks exploit one of the fundamental design principles of the Internet architecture, that any device with an IP address can send packets to any other device with an IP address. This open exchange of information is precisely what has facilitated the explosion of e-commerce and other online services, allowing businesses to receive traffic from new customers all over the world. Yet it is also what enables attackers to generate excessive quantities of unsolicited traffic, and to have it dutifully forwarded to the target. Devices face an intrinsic tension between the utility of processing traffic from unfamiliar endpoints and the costs of doing so.

A wide variety of mitigations have been designed and deployed to combat this threat: various approaches to detecting and filtering out bad packets; additional capacity added to process the bad traffic along with the good; and some attempts to modify network protocols to build in more DoS resilience. While these efforts serve as valuable stop-gap measures their costs are poorly understood. Traffic filtering can be unreliable and add delays, over-provisioning requires monetary investment, and protocol modifications often add more complexity and communication to the network than they alleviate. Mitigations designed for older low-rate floods also remain widely deployed despite lacking empirical knowledge about how they behave under modern attacks. On the whole, existing mitigations fail to address the underlying resource imbalances that facilitate volumetric DoS attacks, instead imposing additional complexity and resource demands on the very devices they aim to protect. This thesis presents a novel approach to reevaluating and redistributing the costs of mitigation, shifting them away from bottleneck resources towards the network edge and onto attackers themselves.

At a high level, our approach begins by re-framing DoS mitigation as a performance optimization problem. Rather than attempting to prevent denial-of-service outright, we seek to maximize the Quality of Service that clients are able to receive. This entails minimizing both the damage an attack can cause, and the overhead a mitigation imposes *outside* periods of attack. Our core strategy for achieving these goals is to shift resource bottlenecks in the network, not only loosening bottlenecks at legitimate devices but also *squeezing* existing bottlenecks at attackers. We aim to drop bad packets as early as possible using as few resources as possible, and/or to limit the ability of attackers to generate large volumes of unsolicited traffic in the first place.

We observe that the vast majority of volumetric attacks rely on a small handful of similar packet types which are easy to generate, relatively expensive to receive, and widely expected to be received from unfamiliar sources. While ordinary clients generate exactly the same types of packets, they do so many orders of magnitude less frequently than attackers. We can leverage this natural asymmetry by making such packets marginally more difficult to generate, causing a large rate-limiting effect for attackers without significantly impacting the performance of others. Taking this a step further, senders can include a self-contained proof of resource expenditure in their packets which can then be verified by receivers, in a concept we call **Trustworthy Independent Packets** (TIPs). Assuming proofs cannot be forged, attackers have no possible method to generate large quantities of valid proof-bearing packets. Either they must reduce their rate/volume of attack as time or other resources are diverted to generating proofs, or else send packets without proofs which can be efficiently dropped once they reach a verifier. This allows us to bootstrap trust between unfamiliar endpoints using a single packet, without relying on prior state or requiring prior/further communication.

As a first proof-of-concept to test this approach, we implement two novel mitigations against the ubiquitous TCP SYN flood attack. We assume attackers are rate-limited by one of two primary resource bottlenecks – for some this will be outgoing bandwidth, but for most it is CPU cycles. To limit bandwidth-constrained attackers we present SYN Padding (§4.2.2), an extremely simple mitigation that increases the minimum length of TCP SYN packets, forcing attackers to send fewer, longer packets. Here the proof of bandwidth expenditure is implicit in the packet's length, and verification is as simple as dropping any packets that are too short. This is largely a toy example designed to illustrate the basic concepts of TIPs, but it proves highly effective when per-packet (rather than per-bit) costs of an attack are dominant, and also sheds light on a general link between packet length and DoS resilience. Our second mitigation, SYN-PoW (§4.2.3), adds a miniature proof-of-work (PoW) to TCP SYN packets in order to rate-limit CPU-bound attackers. The difficulty of PoWs is tunable, with higher difficulty providing a greater reduction of attack volume in exchange for increased overhead at legitimate clients. For both mitigations verification can be performed anywhere in the network, not just at the destination, enabling packets with insufficient proofs to be dropped as early as the first hop.

We implement both mitigations as extended Berkeley Packet Filter (eBPF) programs which are efficient and portable, allowing us to deploy on most modern hardware in a matter of seconds, without modifying any kernel code. We provide copies of our source code for these implementations along with the rest of our research data [3], to facilitate further experimentation and real-world deployment. In order to evaluate these proposed mitigations, we conduct an extensive set of controlled experiments in a physical network testbed. We compare them against each other and against the common SYN Cookies mitigation, evaluating their efficacy at mitigating attacks and the overhead they incur outside periods of attack, and how both these measurements are influenced by the volume of attack, choice of application, topology of the network, and other variables. These experiments demonstrate that our SYN Padding and SYN PoW mitigations both incur negligible overhead at legitimate clients, and both provide extremely high efficacy at reducing attack impact in certain (realistic) contexts.

Both mitigations can be tuned to increase efficacy at the cost of overhead. For SYN Padding this is limited by the maximum TCP header length, allowing us to at most double the length of packets (at most halving the volume of attack). SYN PoW offers far more flexibilility, as it is bounded by subtleties of implementation rather than immutable aspects of the protocol's design. Our eBPF-based implementation is limited to one million CPU instructions worth of work per packet, while a direct in-kernel implementation would be bounded only by the nonce length (32 bits), allowing for proofs representing up to $2^{32} - 1$ average hash function iterations per packet. Padding- and PoW-based mitigations may both be applicable in other protocols as well, and if incorporated through the initial design process we could fully optimize the efficacy/overhead trade-off with no barriers to tunability.

Our experiments also show that SYN Cookies fail to protect against high-volume floods. While they were not explicitly designed with this modern threat in mind, this deficiency is not immediately obvious. Fortunately they do not exhibit significant overhead in our experiments either, but it is important to emphasize that while SYN Cookies may be a valuable tool for DoS mitigation they alone are insufficient against volumetric attacks. The mitigations we propose have the advantage of portable verification, dislocated from the server, which allows them to drop bad packets closer to their source and scale to protect against larger floods.

The metrics and methodologies we use to conduct our experiments constitute a significant component of this research in and of themselves. They are carefully designed to address common issues we have observed across prior work. First, we focus on client-side application-layer indicators of client Quality of Service, rather than more subjective low-level metrics like throughput or latency. Second, we separate a mitigation's *efficacy* at reducing an attack's threat from its *overhead* outside periods of attack, and discuss how to weigh those two key metrics against one another to determine whether a mitigation is truly worth deploying in a given context. Third, we use a physical network testbed to ensure realistic resource bottlenecks, and measure how changes to myriad variables of our network topology influence baseline performance, attack impact, and mitigation utility. This measurement framework has broad applicability beyond the evaluation of our mitigations – it can also be used to compare the DoS vulnerability/resilience of arbitrary network protocols, internet architectures, network topologies, operating systems, applications, etc.

In summary, we argue that the threat of volumetric DoS attacks must be reexamined from an architectural perspective. Attackers disrupt essential services with such ease, frequency, and impunity that the problem has come to undermine the Internet's most basic functionalities. By re-designing key packet types we can make high-volume floods impossible to generate, and/or trivial to detect and drop near their source. By leveraging natural differences in behavior and resource availability between clients and attackers, we can do so without jeopardizing baseline performance.

1.1 Structure of Remaining Chapters

The remainder of this thesis is structured as follows. After restating our key contributions to conclude this chapter we next define the specific problems we are attempting to solve in Chapter §2. We then present background information and discuss related work in Chapter §3, including a more formal definition of the broader DoS problem, taxonomies of known DoS attacks, existing approaches to mitigation, and techniques for mitigation measurement and evaluation. We also provide technical background and a brief history on Proof-of-Work systems to inform discussion of our SYN PoW mitigation. Chapter §4 then provides further detail on our mitigation designs, including general considerations for our TIPs approach as well as specific implementation details for our SYN Padding and SYN PoW mitigations in TCP. In order to evaluate these mitigation designs, we have developed a rigorous experimental methodology and robust suite of metrics, which are presented in Chapter §5. There we also discuss the testbed on which our experiments are performed. Details of those experiments and our analysis of the results are presented in Chapter §6. We then discuss limitations to this work as well as potential extensions, future work and other miscellaneous considerations in Chapter §7 before finally concluding in Chapter §8.

1.2 Key Contributions

The primary contributions of this thesis are as follows:

- We re-frame the broad domain of volumetric DoS mitigation as a more precise and familiar problem of performance optimization.

- We explore the intrinsic connection between packet length and DoS resilience, and show how padding can facilitate a trade-off between attack volume (packets-per-second) and attack rate (bits-per-second).

- We illustrate how adding miniature proofs-of-work to certain packets can significantly rate-limit attackers and enable distributed, self-contained detection of spurious packets that minimizes overhead at vulnerable devices.

- We design and implement both padding- and PoW-based mitigations against the ubiquitous TCP SYN flood, which comply with current standards and can be easily deployed on most modern systems without altering the kernel.

- We define an experimental methodology and set of metrics which can be used to evaluate arbitrary changes in network protocols, architectures, and topologies.

Chapter 2

Problem Statement

The current Internet Architecture is inherently vulnerable to volumetric Denial of Service attacks. Those who wish to offer publicly accessible services face a tension between their desire to receive traffic from new users and the resources that must be consumed or reserved to handle that traffic. A sufficiently large quantity of individually harmless packets will overwhelm even the most capable receivers, forcing them to drop packets or exhaust some other critical resource that disrupts service for legitimate clients. The magnitude of this problem has grown exponentially in the past decade, with attackers leveraging the expansion of global computing power in a sinister shadow of Moore's Law [4], [5]. This is a threat to the Internet's basic duty of packet delivery that demands remediation.

Imagine a server is an ice cream parlor that offers free samples to its customers. Attackers take a sample, immediately re-enter the line, take another, and repeat endlessly without ever buying a thing. On the Internet this problem is compounded by the fact that many attackers are capable of spoofing their addresses – masking their identity – allowing them to evade detection and effectively hold thousands of places in line at once. Not only does the store run out of ice cream to serve its paying customers, but those customers can rarely even make it in the door. Fake customers may even start to block streets as they flock to the shop, preventing other nearby businesses from serving their customers as well.

The problem we face today is akin to an entire nation waiting in line for a single store. Volumetric DoS attacks first crossed the 1 Tb/s threshold in the infamous 2016 Mirai attacks targeting DNS provider Dyn among others [6], and less than one year later Google observed a 2.54 Tb/s attack [7]. The frequency of attacks is also extreme and seemingly ever-increasing – in Q1 of 2022 alone Kaspersky observed 91,052 individual attack events, surpassing the previous quarter by 1.5x and surpassing Q1 of 2021 by 4.5x [8]. Microsoft's Azure cloud platform mitigated over 520,000 unique attack events in 2022, with a minimum of 680 attacks per day[9]. Rates may fluctuate with daily, weekly, or seasonal cycles (as does all Internet traffic), and they may dip temporarily as bugs are patched and bad actors are apprehended by law enforcement, but the long-term trend appears to be towards volumetric attacks becoming exponentially more severe and more common.

As society becomes increasingly reliant on networked services, the potential ramifications of these attacks are becoming ever more dire. The Internet is now an essential tool for communication, commerce, entertainment, education, governance, and healthcare. DoS attacks even threaten national security – the US government publishes guidance for federal agencies on understanding and responding to them [10], [11]. It is also a signatory of the Common Criteria Recognition Arrangement (CCRA), a broadly international agreement based on the Common Criteria for Information Technology Security Evaluation (CC), which includes multiple provisions that explicitly mention Denial-of-Service [12]. Simply put, DoS attacks threaten to disconnect people from one another, and that concerns everyone.

To a large extent the recent increase in attack volume and frequency can be attributed to the proliferation of smart devices that constitute the so-called Internet-of-Things (IoT). Billions of these devices have been connected to the Internet: security cameras, digital video recorders, lightbulbs, toasters, and all manner of other household devices. These devices are typically manufactured by companies that specialize in making those household devices, and that have little if any experience developing secure distributed systems. This combined with a lack of regulation in the space has resulted in devices with egregious vulnerabilities, like using common default passwords and leaving telnet and other vulnerable ports open for no reason. Some devices even lack the capability to receive security patches if and when these vulnerabilities are discovered, and without regulation mandating device recalls or consumer notification vulnerable devices are likely to remain in use for extended periods of time[13]–[15]. Typically IoT devices have limited computing resources which may preclude them from running standard security protocols like TLS, making them even more difficult to secure [16]. However, we will demonstrate in Chapter §4 that the existence of such resource limitations at individual attacker devices can be used to our advantage in designing volumetric DoS mitigations.

In terms of the actual attack types being launched, there is a very wide variety but with a non-uniform distribution. Looking longitudinally TCP SYN floods have been the single most dominant vector, and according to most sources they still account for approximately 50% of all floods today. In recent years, short- and zero-payload UDP floods have become increasingly common, accounting for most of the remaining attacks and by some accounts even surpassing SYN floods. In the first two quarters of 2022 Cloudflare reported that SYN floods accounted for 57.4% and 53.6% of observed “network attacks” [17], [18], while

Kaspersky reported them as 22.37% and 20.25% of observed “DDoS attacks” for the same periods with UDP flooding instead taking first place[8], [19]. Microsoft Azure reported 63% of all DoS attacks they observed in 2022 were TCP flooding, though that figure includes ACK floods and other TCP floods in addition to SYNs, and a more detailed breakdown was not provided. Beyond these are a multitude of other vectors – several sources (including those listed above) have noted a recent uptick in HTTP and certain other application-layer attacks, but not nearly significant enough to compete with SYN and UDP floods.

We observe that these two most common attack vectors both rely on packet types that devices expect to receive from new, untrusted sources. DNS reflection/amplification attacks, which are commonly ranked as the third most common vector, also fall into this category. Herein lies our central motivating problem: a willingness to handle such packets is fundamental to the Internet’s operation, but the act of handling them can render it inoperable.

The cost of handling a packet varies depending on its contents and the roles of the device acting on it. A server receiving a TCP SYN from new IP address will allocate space in a limited segment of memory to store connection information and expend computational resources constructing a SYN-ACK in response. A UDP packet from a new source will prompt a server to check if it has a service running on the specified port which takes a small but non-zero amount of work – if not it may expend some effort to generate an error in response, or if so it may be prompted to transmit a large file or perform some other expensive action. Forwarding each of these packets and the responses they elicit through the network also incurs costs at every hop, adding CPU overhead to switches, filling queues, and forcing legitimate traffic to be dropped. Extremely small TCP SYNs and zero-payload UDP packets magnify this per-packet damage. Assuming a fixed bit-rate of incoming data, smaller packets mean more packets per second and more packet processing overhead. At a minimum, every device the packet transits will need to perform a routing lookup, decrement the time-to-live (TTL) field, and update the IP checksum. This overhead is near-constant for every IP packet received, regardless of size.

There is an inherent threat posed by any packet type that is a) cheap to generate, b) expensive to receive, and c) necessary to receive from untrusted sources. We propose to address this problem by making such packets more expensive to generate, and by aiding receivers in dropping them earlier and more efficiently. If every visitor to our metaphorical ice cream shop had to pay just one cent before they could get in line, the actual customers would happily oblige while the millions of fakes would have to pay a combined fortune to continue holding their places.

In the following Chapter (§3) we provide further background information on the DoS problem more broadly and existing approaches to addressing it, before presenting our own novel mitigation designs in Chapter §4.

Chapter 3

Background

This thesis builds on a rich background of prior research, on Denial-of-Service and several other topics. Before presenting our mitigations, metrics, and experimental results, we must first frame the problem in the context of this existing work. We begin with more formal definitions and taxonomies used to classify DoS attacks (§3.1). Through these classifications we further define the scope of the problem we aim to address, outlining which categories of attacks our mitigations target and why. We then discuss current approaches to mitigating the threats of DoS attacks, including some shortcomings of each (§3.2). After that we discuss prior efforts to evaluate DoS mitigations, including metric definitions and experimental methodologies (§3.3). Finally we provide technical background on Proof-of-Work (PoW), which is used in the SYN-PoW mitigation we present in Section §4.2.3, and a brief history of its application to the DoS problem (§3.4).

3.1 DoS Attack Classification

The term "Denial-of-Service" is a large umbrella that encompasses many drastically different types of attack. An early formal definition was provided by Gligor in 1984 [20]:

> “A group of authorized users of a specified service is said to deny service to another group of authorized users if the former group makes the specified service unavailable to the latter group for a period of time which exceeds the intended service [maximum waiting time].”

This remains a fairly accurate generalization of the problem, despite being written nearly 40 years ago and in the context of operating systems rather than networks. In addition to outright *denial* of service in which some resource is made completely unavailable, we are also concerned with *degradation* of service, in which some metric defining the *quality* of service (QoS) falls below acceptable levels. We hereafter use the abbreviation “DoS” to include both denial and degradation of service. DoS attacks can involve preventing new connections from being established as well as disrupting, terminating, or even hijacking ongoing connections.

In 1993, Needham [21] considered network Denial of Service more specifically, dividing the problem into attacks on the server, the client, and the network itself. These remain the three primary target types, but in reality a single attack will often have some impact at multiple locations in the network. Each attack packet will consume some resources at its destination as well as every hop along the way, and any response packets it elicits will do the same. There is further complexity within each device as well, as attacks may target a variety of different resources at multiple layers in the protocol stack.

The diversity of DoS attacks makes it challenging to reason systematically about defense. They cause different types of disruption to different targets using different mechanisms. To begin teasing apart this problem space further we refer to the taxonomy of DoS attacks and mitigations presented by Mirkovic and Reiher in 2004 [22], which remains the most thorough to date. Though the DoS landscape has changed significantly in the nearly 20 years since its publication, their categorizations remain a useful starting place for building common language.

At a high-level, they classify attacks according to the eight factors listed below. In the remainder of this section we give a brief overview of each factor and discuss how it does or does not pertain to this work. We have re-ordered them from roughly most to least relevant, and labeled each with the subsection number in which it is discussed:

1. (§3.1.1) **Exploited Weakness to Deny Service**: (Semantic / Brute-Force)
2. (§3.1.1) **Victim Type**: (Application / Host / Resource / Network / Infrastructure)
3. (§3.1.1) **Attack Rate Dynamics**: (Constant / Variable)
4. (§3.1.2) **Source Address Validity**: (Spoofed / Valid)
5. (§3.1.3) **Possibility of Characterization**: (Characterizable / Non-Characterizable)
6. (§3.1.3) **Impact on the Victim**: (Disruptive / Degrading)
7. (§3.1.4) **Persistence of Agent Set**: (Constant / Variable)
8. (§3.1.4) **Degree of Automation**: (Manual / Semi-Automatic / Automatic)

3.1.1 Volumetric vs. Targeted Attacks

We begin our own classification with what [22] refers to as the **Exploited Weakness**, dividing attacks into two primary categories: **volumetric** (brute-force) and **targeted** (semantic). In **volumetric** attacks, attackers send large floods of traffic to exhaust receivers and/or the intermediary network infrastructure by brute force. The higher the volume of traffic, the more damaging we expect the attack to be. Targeted attacks exploit more precise vulnerabilities in protocol design or implementation to degrade service, typically with a much smaller amount of traffic. Some targeted attacks may even *require* a specific low rate, such as those that exploit TCP retransmission timeouts [23].

We constrain our scope primarily to volumetric attacks, for two main reasons. First, they are the predominant threat on the Internet today, as detailed in Chapter §2. Second, they offer *some* constant trait by which we can begin to compare them: the attack's volume. The space of targeted attacks by contrast is completely boundless and unstructured in a way that makes systematic analysis extremely difficult. Yet this distinction between targeted and volumetric attacks is not as clear as it may seem. In some cases the same packets used to launch a targeted attack can also be used to launch a volumetric attack. To accommodate this ambiguity we expand our scope slightly: in addition to measuring and mitigating volumetric attacks, we are also interested in the thresholds at which an attack transitions from exhibiting targeted to volumetric behavior. If a mitigation is only designed for one of the two attack variants, that threshold defines a bound on the mitigation's efficacy.

This brings us to Mirkovic and Reiher's category of **Victim Type**. Typically, the scale at which an attack becomes volumetric is the point at which the primary bottleneck transitions from one of an application/host resource to one of a network/infrastructure resource. TCP SYN floods offer a prime example of this transitional behavior. At relatively low volumes they operate by exhausting the limited connection slots of a TCP server, but at sufficiently high volumes they also congest the network and force clients' connection requests to be dropped before they ever reach the server. TCP SYN cookies work well to mitigate lower-rate attacks by delaying state allocation until the server can confirm the SYN's source address, preventing address spoofers from tying up connection table resources, but as we will demonstrate in Chapter §6 their efficacy declines steadily as the attack rate increases.

Identifying which victim type presents the primary bottleneck can help us identify why an attack is effective, but most attacks will impact multiple victim types to some extent, and what we really care about is the combined effects on *all* parts of a system that an attack traverses. The real question is what end users actually experience. In terms of the Disruptive vs. Degrading framing for **Impact on the Victim** in [22], we assume that the attacker's goal is generally to cause as much damage as possible given the resources they have available. Quantifying this damage is an extremely complex task to which we devote Chapter §5. Our metrics build on prior work also led by Mirkovic, as discussed further in Section §3.3.

As we are limiting our scope to volumetric attacks, our consideration of **Attack Rate Dynamics** is slightly different from that of Mirkovic and Reiher. They divide attackers into those that send at a constant rate and those that send at a variable rate, either in a way that gradually increases or randomly fluctuates. In the measurement framework we present in Chapter §5, the rate of an attack is *the* key variable that determines its threat. While we are extremely interested in changes to the attack rate, we want to measure the effects of those changes in a controlled way. Rather than using variable-rate attackers directly, our model measures many different constant rates of attack through independent trials. This allows us to more accurately quantify the tipping points at which resource bottlenecks shift, and at which attacks transition between targeted and volumetric forms.

We've found that high-volume attacks generally take effect extremely quickly, shifting a system from one steady state to another within a matter of a few seconds or less after they are launched. There is rarely a significant ramp-up period, but often a long ramp-down after the attack ends, while its side-effects continue to propagate through the network. It is perhaps for these reasons that burst attacks are commonly observed in the wild – if an attacker can shut down a site for 10 seconds by sending a 1 second burst, that allows them to spend the other 9 seconds attacking other targets, or staying quiet to mask their activity. Though our experiments use constant-rate attacks for the sake of simplicity and consistency, each one can be thought of as measuring the effect of a single burst.

3.1.2 Address Spoofing

The question of **Source Address Validity** is a very significant concern as the IPv4 and IPv6 protocols lack a strong authentication mechanism, making it trivially easy for many hosts to spoof their source addresses. This enables a single attacker to masquerade as countless others, requesting excessive resources that it doesn't want and will never actually receive. Address-based filtering and rate-limiting approaches become unreliable. Spoofing also enables reflection attacks, wherein attackers use the source address of their victim to send widespread requests for content. That content is then delivered to the victim, overwhelming it with unwanted traffic.

Address spoofing can be mitigated effectively by would-be attackers' ISPs with the use of source address validation, but despite long-established best practices and the community's best efforts deployment of that technology appears to have stagnated. In 2019, Luckie et al. found that out of 5,178 autonomous systems tested, 31.5% had at least one address prefix that was spoofable [24]. This state of affairs is essentially unchanged from what Beverly et al. observed 10 years prior: that 31% of clients tested could successfully send traffic with arbitrarily spoofed source IP addresses, while 77% of the rest could still spoof within their own /24 subnet [25]. These gaps in security weaken the entire network – ingress filtering can only effectively be performed at the edge, so once a spoofed packet leaves its home network it becomes extraordinarily more difficult to detect.

Publicly available source code for the Mirai botnet reveals that its bots will spoof their source addresses when possible[26]. Mirai has facilitated several high-profile attacks, and variants of it are still widely used to this day[6], [13], [27], [28]. Even when an attack is launched from millions of devices, its impact can still be magnified further by having each of those devices act as millions more. Data from Microsoft Azure confirms this, as in 2022 they reported that 53% of UDP flooding traffic was spoofed, though how they were able to make this characterization is unclear.

The mitigations we propose in Chapter §4 operate under the assumption that address spoofing will remain a pervasive threat for the foreseeable future. They aim to provide an alternative to source address validation that is capable of allocating resources fairly among anonymous devices, without the need for authentication and without relying on cooperation from the attacker's first-hop service provider.

3.1.3 Packet Types

One important factor left under-specified in [22] is the type of packet being used to launch an attack – what protocol is it a part of; what layer does that protocol operate at; does it conform to or deviate from the protocol standard; is it establishing a connection, requesting data, closing a connection, modifying a connection, sending/eliciting error/debug messages, etc.? It is understandable why they would have excluded such questions from their taxonomy, as there is simply too much variety to categorize in a way that is both concise and meaningful. That would effectively amount to a taxonomy of all network protocols, of which there are hundreds, with over 9000 RFCs specifying their operation and other considerations for their design and use.[29]

As a result, the attacker's choice of packet type will have a tremendous influence over their **Impact on the Victim**. Different packet types will consume different resources at different locations, which impact different client applications in different ways. Some attacks may use packets from the same protocol they are attacking to exhaust protocol-specific resources, while others drain system-level resources to hinder services that rely on entirely different protocols. We are interested in all manner of potentially dangerous packet types but focus our mitigation design and analysis around those that pose the greatest threat today, as discussed in Chapter §2.

Packet type is also relevant to the **Possibility of Characterization**, whether or not the attack packets can be reliably distinguished from legitimate ones. Certain packet types are significantly easier to characterize than others. Attack traffic that does not conform to an existing protocol standard is trivial for devices to detect and drop. At the other extreme, many attacks rely on packet types that individually are completely benign and only cause damage in large volumes, making their characterization impossible. Others fall somewhere in-between, with receivers able to determine they are malicious only after expending some significant amount of time or other resources to analyze their contents.

Our primary interest is the mitigation of attacks that are non-characterizable, as they account for the majority of the threat on the modern Internet. Floods of seemingly normal traffic are less likely to be detected and dropped at any given hop, and therefore more likely to consume the maximal amount of resources at their destination and in the intervening network. The mitigations we present in Chapter §4 operate by adding some additional information to packets that assists with characterization – adding the information requires attackers to reduce their flooding rate, and not adding it allows receivers to detect and drop their packets.

3.1.4 Other

Among the other categories, the **Degree of Automation** is of little concern – we assume bad actors will find some way to infiltrate devices and use them to generate attack traffic. We recognize the importance of efforts to identify those personally responsible, and to intercept communications between them and the devices they command, but our threat model assumes that some significant amount of malicious traffic is inevitable in the Internet. As new hardware and software is developed, new vulnerabilities are created that can and will be exploited to launch attacks. Our concern is how to measure and minimize the damage those attacks are capable of causing.

The lack of ability to characterize traffic or validate source addresses also makes the **Persistence of the Agent Set** somewhat irrelevant. If we cannot tell where the attack is coming from, or even which packets are part of an attack, then it doesn't matter much if the set of source devices is changing. Our experiments use a small, constant set of attackers spoofing a wide range of addresses. From any given receiver's perspective this is functionally equivalent to a large set of attackers. What matters most is the volume of traffic arriving, not how far it has traveled or from where.

3.2 Existing DoS Mitigations

To overview existing DoS mitigation approaches, we again begin with the scaffolding provided by Mirkovic and Reiher's taxonomy, summarized as follows:

- **Activity Level** (Preventive / Reactive)
- **Cooperation Degree** (Autonomous / Cooperative / Interdependent)
- **Deployment Location** (Victim Network / Intermediate Network / Source Network)

Within this context we are primarily interested in designing preventive approaches – modifications to network protocols and Internet architectures that make them fundamentally more resilient to attack. We also want mitigations to function autonomously. Certain types of cooperation are beneficial, but we want to avoid those that add significant communication overhead. Deployment location is a complex question but generally we aim to move the burden of mitigation away from targeted bottleneck resources, towards the network edge and ideally onto attackers themselves. This may involve certain functionality scattered throughout the network, making it more difficult for attackers to generate damaging floods, and aiding servers and routers in determining which traffic to drop.

For our own classification we divide existing mitigation approaches into three main categories: traffic filtering (§3.2.1), over-provisioning (§3.2.2), and protocol modifications (§3.2.3). A summary of the three approaches is provided in §3.2.4.

3.2.1 Traffic Filtering

Many mitigation approaches rely on the ability to detect malicious traffic and drop it, ideally as close to the network edge as possible. Content Distribution Network (CDN) operators like Cloudflare and Akamai provide such traffic filtering services for a price. The exact methods of these Mitigation Service Providers (MSPs) are proprietary, but involve some combination of IP address blacklisting, rate limiting, and attack pattern detection, sometimes using machine learning.[30]

In Mirkovic & Reiher's terminology these approaches are reactive, generally taking at least a few seconds to take effect after an attack is launched (at which point significant damage may already be done. They may be autonomous, implemented solely by the mitigation service provider, but can and likely often do benefit from cooperation among different MSPs and ISPs. They are primarily deployed in the intermediate network, though these same MSPs also typically provide content distribution services which means they assume the role of source network as well.

As discussed above in Section §3.1, our primary concern is mitigating attacks that are *non-characterizable*, against which such traffic filtering strategies are inherently ineffective.

When attackers spoof their source addresses those addresses can't be used as a basis to drop or rate-limit traffic. When individual packets are standards-compliant, identical in structure to those sent by legitimate clients, and therefore expected by the server, there is no reliable way to distinguish good packets from bad. A flooding attack may look identical to a genuine spike in demand for some service, in terms of both the traffic received and the impact on user experience. Attackers may even spoof the addresses of legitimate clients, intentionally or accidentally, making their packets completely identical. As such, traffic filtering strategies are bound to suffer from some number of false positives, wherein requests from legitimate clients are mistaken for attack traffic and dropped. In such cases the entity that is supposed to be mitigating a DoS attack effectively launches one instead.

It should also be noted that filtering is an active mitigation which adds computational overhead to analyze each packet, and those which measure traffic patterns over extended periods of time require significant storage in the network as well. Procuring and operating those services at Internet-scale is expensive, and they are usually[1] not provided for free. These costs are paid by service providers and passed on to consumers, making the Internet more expensive to use. They also reduce competition by pricing out smaller businesses that can't afford paid mitigation services and therefore can't provide adequate service quality to their customers.

Additionally, any mitigation strategy that favors one type of packet over another raises potential ethical concerns. Some filtering strategies employ Deep-Packet Inspection (DPI) to act on higher-layer packet headers and even data payloads [31]–[33]. While the additional

[1]Cloudflare's Project Galileo offers organizations "in the arts, human rights, civil society, journalism, or democracy" an opportunity to apply for free DoS protection services. Other companies may offer similar free or reduced-cost mitigation for select services.

information may be useful in characterizing malicious traffic, it may also jeopardize user privacy and network neutrality. DeRose conducted a literature review of this relationship between DPI and net neutrality in 2010 [34], concluding that "DPI is at odds with a free Internet" and "it also gives ISPs the means to threaten net neutrality." Any system that employs machine learning (for DoS mitigation or any other application) must be careful to avoid bias and unfairness among different users, as show in Mehrabi et al's 2021 survey on the subject [35]. While a large amount of research is being conducted on the development of explainable artificial intelligence systems, many of them remain effectively black boxes, the operators of which may not even know why any given packet is chosen to be dropped. Došilović et al. survey the wide extent of this problem as well as potential approaches to addressing it in [36].

3.2.2 Over-Provisioning

When filtering fails or isn't possible, the backup plan for MSPs is to simply process the flood of junk traffic along with the trickle of real requests. Traffic is routed through the MSP's private network and load-balanced across numerous replica servers hosting copies of the targeted content/service. This synergizes with their role as a CDN – even without concern of attack, performance can be improved by caching replicas of content at multiple devices, ideally edge devices near the clients most likely to want that content. As mentioned, a flooding attack is often indistinguishable from a legitimate demand spike, and so strategies for efficiently delivering popular content are naturally similar to strategies for mitigating floods.

This strategy has proven remarkably effective, allowing providers to maintain service during floods reaching hundreds of millions of packets and trillions of bits per second.[37] However, the monetary costs this approach imposes are even worse than those of traffic filtering. Deploying enough network capacity to handle a flood that has a volume X times the normal demand means buying X times as many servers/routers/cables, which then require X times the cost to operate and maintain. Economies of scale may help reduce these costs somewhat for large MSPs, but generally speaking larger floods will always be more expensive to mitigate. Again, these costs are borne by service providers and therefore by their customers. It is an approach based on deliberate waste – in a sense excess resources are allocated explicitly for the use of attackers. While the strategies we propose in Chapter §4 also involve some amount of waste, it is carefully minimized and largely confined to the attacking devices themselves, placing little to no cost on service providers.

3.2.3 Protocol Modifications

Our third category is the broadest, encompassing any mitigations that modify the basic operation of network protocols. Such approaches have perhaps the greatest potential for both benefit and harm. Ideally we should strive to design more DoS-resilient protocols from the ground up, to make it difficult for even the most clever and well-resourced attackers to cause harm, without needing to rely on expensive or invasive mitigation services. Yet this is much easier said than done – network protocols are extraordinarily complex, and it is not clear what makes one more or less resilient than another. A mitigation that blocks one attack may collapse under another, and the same attack can have drastically different effects when launched against two sets of devices with different hardware, software, or connecting topologies, even if they deploy the same mitigation.

One common type of protocol modification is the inclusion of a client puzzle. Perhaps the most notable examples are SYN Cookies and SYN Cache in TCP [38], wherein upon receipt of a SYN the server encodes vital information as a “state cookie” in a portion of the SYN-ACK rather than storing it locally. The client echoes the information back in its ACK, from which the server reconstructs the necessary data. With SYN Cookies the server discards all connection state after sending its SYN-ACK, and for SYN Cache it stores only a portion of the usual data. This ensures that connections are not reserved for address-spoofing clients, which are unable to receive the SYN-ACKs and the state cookies they contain. Cookies are signed by the server to prevent ACK forgery. This process remains a non-standard yet widely implemented aspect of TCP. SCTP standardizes the approach by replacing TCP’s 3-way handshake with a 4-way initialization that includes a similar cookie [39].

Generally, we can model such mechanisms as the following 4-step process:

1. Client requests a puzzle

2. Server generates and sends a puzzle

3. Client computes and sends a solution

4. Server validates the solution

In the case of SYN Cookies and SYN Cache, “computing a solution” in step 3 is effortless – the client receives a state cookie invisibly encoded in the sequence number field and echoes it back in the acknowledgement number field[2], exactly as in normal TCP [40]. This allows

[2]Some implementations make use of the timestamp option fields to encode additional information.

the puzzle implementation to be server-specific since a client doesn't even need to know it is participating. Not all implementations are equivalent though – the sequence number field is clearly not large enough to hold all the information from the SYN, so the server must choose which data is most essential and discard the rest. This has resulted in implementations that preclude the use of certain TCP options, as the server discards information that would indicate whether the client supports those options [41].

In other puzzle systems the client may be presented with a more complex challenge that is expected to take some amount of time or other resource(s) to solve. At the application layer, many websites rely on mitigations like Cloudflare's Browser Integrity Check [42] which adds multiple seconds of latency to an HTTP request, as well as human-level challenges like CAPTCHA [43] and its successor reCAPTCHA [44] which reportedly take the average user 32 seconds to complete, totaling an estimated *500 years per day* of wasted time [45]. The benefits of these mitigations are difficult to quantify, but it is hard to imagine they could justify such extreme overhead.

Even when well-implemented, such puzzle-based mitigations add overhead at every point in the system they aim to protect. Servers must generate puzzles and validate solutions, clients must solve puzzles, and the network must carry the puzzles as well as the requests and solutions. Volumetric floods multiply this load at the server and in the network, turning the puzzle generation and distribution services into DoS attack targets themselves. A 2004 draft paper from Beal and Shepard provides an excellent analysis of exactly how this issue manifests and how to design effective puzzle systems for DoS de-amplification around it [46]. While SYN Cookies, SYN Cache, and other similar client puzzle mechanisms were not designed to mitigate today's ultra-high-volume floods, they often remain deployed when such floods happen to arrive. Our TIPs eliminate steps 1 and 2 above by generating puzzles on-demand at the client, and TIP validation can be offloaded to edge devices to further reduce the server's load.

While not directly related, we also note that some prior work in queuing theory and congestion control also provides useful background. Volumetric attacks are similar to if not indistinguishable from spikes in demand that cause ordinary congestion, and so designing protocols to be more fair and performant in the face of legitimate congestion is much the same as designing them to resist DoS floods. As we discuss in Section §4.2.3 below, one extension of our SYN-PoW proposal is essentially a form of *enforceable* congestion control in which greedy endpoints are rendered physically incapable of generating large quantities of valid packets. Congestion control is a rich area of research with literature too vast to enumerate, but we highlight a 1985 paper by Stidham [47] which builds on earlier work by Naor [48] to theoretically analyze a cost-based model for optimizing admission to network queues. Though the Internet has changed dramatically since these papers were published, the fundamental mathematics they use have not. They discuss differences between static and dynamic control models, as well as the inherent tension between optimizing for individual performance versus designing a "socially optimal" system which aims to maximize performance and fairness across all users.

3.2.4 Summary

To summarize, we classify existing mitigations as either **traffic filtering**, **over-provisioning**, or **protocol modifications**. Traffic filtering works well for attacks consisting of easily characterizable traffic, but the most damaging attacks today often use packet types that are indistinguishable from legitimate ones. Filtering also fails to protect against any novel zero-day vulnerabilities (until new filtering rules can be established for them), and requires a depth of packet analysis that may negatively impact both performance and user privacy. Over-provisioning is commonly employed by large CDN operators that also provide DoS mitigation services, using their excess infrastructure to simply handle the flood and process the traffic, primarily by hosting replicas of vulnerable services on a geographically distributed set of devices. This approach has proven successful at mitigating many large-scale attacks, but processing such large volumes of junk traffic is inherently wasteful, and aquiring such mitigation services is not free. The replication approach is also infeasible for certain types interactive services that require communication with a specific device, and for services handling private data that cannot legally be copied to third-parties.

3.3 DoS Measurement

In the realm of metrics and methodology for DoS Mitigation, we are building primarily on a series of papers led by Jelena Mirkovic, which deal with the accurate measurement of DoS attacks [49]–[51]. The primary takeaway from that research is that different client applications have drastically different requirements in terms of network metrics like loss, latency, throughput, and jitter, and that we should instead prioritize client-side metrics which more directly capture the Quality of Service (QoS) experienced by end users. For instance, real-time audio conversations require low latency but can tolerate fairly high loss rates and low throughput, while streaming video requires high throughput but can tolerate considerable latency by buffering at the client. The same applies to resource metrics like CPU utilization and queue lengths – different applications may have drastically different needs, and it can be difficult to disentangle the ways in which they depend on various system resources and properties of the network.

Accurately quantifying QoS for these applications requires us to actually run them and observe what clients experience. Mirkovic et al. primarily measure QoS in terms of the rate at which clients can complete some application-layer transactions with a server, and measure changes in that rate under various attacks. Their DoS attack metrics provide a solid foundation to build on, but we have found that extending them to the realm of DoS mitigations is surprisingly non-trivial, for two main reasons. First is that deploying a mitigation may incur significant overhead, and we need metrics which capture the impact of that overhead on client QoS *outside* periods of attack. Second is that a mitigation's efficacy and overhead may both be variable, depending not only on the client application but also on the rate of attack, the network topology, and myriad other factors.

Prior work evaluating the efficacy of specific mitigations is surprisingly scarce. Mitigation service providers periodically publish whitepapers with statistics about specific attack events they have observed, but these sources are largely promotional in nature and lack raw data or robust statistical analysis. A handful of prior studies have performed a more rigorous analysis of SYN Cookies in particular, but all suffer from significant limitations – in fact, it is precisely this gap in the literature that originally motivated our work.

First was a brief but relatively thorough analysis from Lemon in 2002, which measured connection completion rates with both SYN-Cache and SYN-Cookies [40]. The compared performance during idle periods and times of attack, but tested on a fairly simple four-device network topology with maximum flood rates of only 15,000 packets/second (15 Kpps), and did not measure a standard TCP implementation as a control. Next came a 2008 study by Smith and Watrawy comparing different OS implementations – this is an important contextual variable to measure, but their analysis failed to consider the mitigation's overhead outside of attack, used only a single low-volume attack rate (80 kB/s), and tested on an unrealistic network topology with only two devices [41]. A 2018 paper by Echevarria et al. measured SYN Cookies' efficacy in the specific context of network-constrained devices with similar issues – no consideration of mitigation overhead, a single slow attack rate of 200 packets/second, and a topology of just three devices connected via a single switch [52]. The most thorough analysis of SYN Cookies we have seen to date is presented by Scholz et al. in a 2020 pre-print [53]. They compare efficacy for different attack rates, client request rates, kernel versions, and even different variants on the mitigation. This sort of cross-context measurement is a large step in the right direction, but relied on a single overly simplistic network topology with one device acting as both client and attacker. Reusing the same device for multiple roles in this way makes it impossible to accurately measure resource utilization. Also relevant is a 2019 paper by Noureddine et al. [54], which proposes and evaluates an approach similar to SYN Cookies with an advanced mechanism for selecting puzzle difficulty. That mechanism may prove relevant for a possible extension to our SYN-PoW mitigation, and their experimental topology is relatively complex, but they too test only meager floods of at most 5,000 packets per second. In our own prior work using a preliminary version of our measurement framework we observed a dramatic change in the behavior of SYN Cookies at the point when flood traffic and response traffic combine to saturate some bottleneck link, and demonstrated that this tipping point depends on multiple contextual variables [55].

Even though SYN Cookies and similar puzzle mechanisms were not designed to mitigate high-volume floods, it is still important to understand how they behave under such attacks, and what specific traffic rates trigger changes in their behavior. A device that deploys them as a precaution against low-volume floods will still have them deployed if/when a high-volume flood arrives, and they may become counter-productive at maintaining availability during such attacks. Generating puzzles for bogus SYNs requires CPU overhead at the server, and (re)transmitting SYN-ACKs in response wastes network resources. Our measurement framework is specifically designed to capture such costs, and our attacker-side mitigation approach is designed to minimize them.

3.4 Proof-of-Work

This section provides a brief overview of Proof-of-Work (PoW) systems broadly, as well as prior efforts to apply them as DoS mitigations. PoW provides a tool for one device (the **prover**) to assert to another (a **verifier**) that it has expended some computational resources, in a way that can be efficiently verified, albeit with some statistical uncertainty. This has clear applications for DoS mitigation – generating spurious traffic requires computational resources, so requiring senders to expend additional CPU cycles with each packet could force them to send less.

Basic operation for constructing such a proof is as follows, and as depicted in the top half of Figure 3.1:

1. Construct some message m

2. Generate a random nonce n

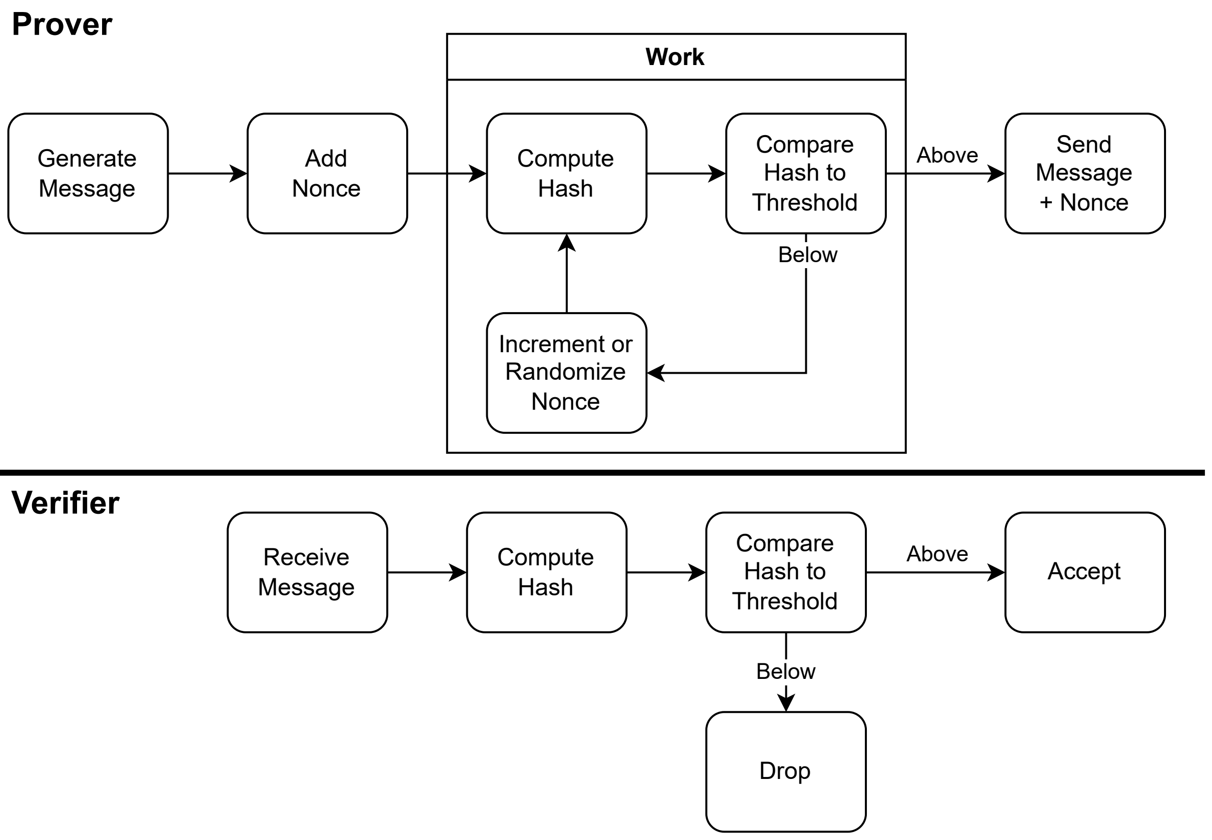

Figure 3.1: Basic operation of a Proof-of-Work (PoW) system. The prover repeatedly changes some portion of its message (the nonce) and recomputes the message hash, until that hash exceeds a given threshold. The verifier performs a single operation of the same hash to confirm whether incoming packets meet the threshold.

3. Compute $h = H(m|n)$ using some well-known one-way hash function H

4. Compare h to some threshold θ

   (a) If $h \geq \theta$: send $m|n$

   (b) If $h < \theta$: increment n and repeat from step 3

The verifier can then compute $h = H(m|n)$ using the same hash function[3], and decide

[3]Note that concatenation is not the only option for combining m and n. Depending on the application it may be possible to merge them in some other way. For example, if m is a network packet, it may contain some existing field that can be used to store the value of n. What matters is that the prover and verifier both use the same hash function and construct their inputs to it in the same way, and that the verifier can interpret the message as the prover intended. Some applications may utilize PoW without any other message passing, in which case m is simply an empty string (such that $m|n = n$.

how to handle the message by comparing h to θ, as shown in the bottom half of Figure 3.1. Selecting an appropriate θ value is a complex and application-specific task. We elaborate on considerations for our own application in Section §4.2.3, but in the simple case we assume all participants have agreed on some standard value in advance. Note that there is no benefit to transmitting the hash itself, since the prover and verifier can each compute it independently. In that paradigm a malicious prover could easily claim a higher hash value than its message actually had, so verifiers would need to take exactly the same steps to compute the real hash to compare against the claimed value. The CPU cycles that provers spend repeatedly generating random nonce values and computing hash functions is what constitutes the "work." The "proof" comes from an assumption that H truly is a secure one-way hash function, a mathematical primitive defined by Naor and Yung in 1989 [56]:

> "The property required from the hash function h is that for a given value x it is computationally hard to find a y such that $h(y) = h(x)$ and $y \neq x$."

In our notation this means that for a given message m, it is provably difficult to find a nonce n such that $H(m|n) \geq \theta$. We assume that it is the goal of all provers, including malicious ones, to generate valid proofs for multiple different messages. In other words, once an attacker has found some m, n pair such that $H(m|n) \geq \theta$ they cannot simply send that pair over and over. They send it once and then must generate a new message m' for which a new nonce n' must be found. This is certainly true for legitimate clients who want to send and receive new information with each message, but it is also true of nearly all malicious clients who seek to consume new resources with each message. Some attackers may still be able to gain an advantage by pre-computing a large number of valid message/nonce pairs and sending them all at once, but as we will discuss in Chapter §4 this approach is sub-optimal for launching high-volume DoS attacks.

Constructing such a hash function relies on additional assumptions and mathematical primitives, the complexities of which are well beyond our scope. A wide array of implementations exist, each of which make trade-offs between security and performance relating to the primitives they build on. Further technical details and a survey of various function designs are provided in [57], [58]. When designing our own PoW-based mitigation in Section §4.2.3 we consider implications these details could have on our system, but to a large extent hash functions are interchangeable, performing the same task in slightly different ways.

We are not the first to recommend such a hash-based proof-of-work system for DoS mitigation. Indeed, the very first proposal for a PoW mechanism was Dwork & Naor's "Pricing via Processing or Combating Junk Mail" [59] which suggested its use for spam email mitigation in 1992. Senders would generate a PoW to include in each message, and mail servers would only deliver messages with valid proofs. Clearly that particular application did not succeed, likely due to reasons discussed in "Proof of Work Proves not to Work."[60] Despite the generality of that paper's title, their analysis is specific to the context of spam email, and the PoW approach failing in that domain should not be seen as an indication that it cannot work elsewhere. Spam email is a very different problem from volumetric DoS attacks at the transport- and network-layers. The most significant difference is the scale at which they operate – the number of packets per second required to overwhelm a switch is many orders of magnitude greater than the number of emails per second required to overwhelm a human recipient. This means the burden on attackers to generate per-packet proofs far exceeds what would be required for per-email proofs, all else being equal.

This enables us to set proof thresholds so low that they negligibly impact legitimate clients while still significantly impeding attackers, whereas the email application cannot provide meaningful protection without incurring unacceptable delays.

Today, PoW is most often discussed as a distributed consensus mechanism for blockchain systems such as Bitcoin [61]. That application has some similarity to DoS mitigation in that the systems are attempting to limit the rate at which participants are able to generate and submit valid blocks, in order to prevent the simultaneous submission of multiple conflicting blocks that would "fork" the chain and break consensus. However, we are not trying to establish consistency because there is no need to guarantee a global ordering among clients. There is not even a need to guarantee ordering of packets from a single client since re-ordering can be performed by TCP where needed. This means that rather than competing with one another to find *the* next valid block, each SYN-PoW client competes only with itself to find *a* valid packet. Other devices finding other packets while it works are entirely independent events that do not invalidate whatever work has been done. Clients are guaranteed to find a valid proof eventually, so while they incur a few CPU cycles of overhead per packet it all goes towards a useful purpose. PoW-based consensus systems are often criticized for their energy consumption [62] but in reality that applies much more to some implementations than others [63], and while we are still proposing a small amount of "wasted" work, the net result is better overall efficiency.

Chapter 4

Designing Trustworthy Independent Packets

The Internet was designed as an open platform through which remote endpoints can freely exchange information. Access to early iterations of the system was restricted to a relatively small set of known participants, and this personal familiarity among users provided some inherent sense of trust. In *The Design Philosophy of the DARPA Internet Protocols* [64], the first of Clark's seven stated goals is that "Internet communication must continue despite loss of networks or gateways," but the primary threats to communication were then seen as hardware failures, poor software implementations, and misconfigurations. As the Internet grew and evolved, securing it against deliberate attack became a greater concern, with an increasingly large and anonymous userbase seeking to access an increasingly valuable and even vital set of services. Yet by this time the architecture had already become too deeply entrenched for a major redesign to be feasible. The core TCP/IP protocols still remain largely unchanged since their inception more than 40 years ago, with most security features instead being added on at higher layers.

This approach is intentional, following from the end-to-end arguments [65]. There is an assumption that security is a luxury not all traffic requires, and that implementing it in the network core would add unnecessary overhead. In reality this is a drastic oversimplification, as "security" can refer to many different properties. We assert that one particular security threat, that of volumetric DoS attacks, is so severe and so ubiquitous that it warrants a more architectural approach to mitigation. There is an inherent tension in the current architecture between the fundamental need for public servers to accept traffic from unknown hosts, and the resources that must be expended to handle that traffic. The ease and impunity with which attackers are able to launch high-volume floods routinely undermines the entire Internet's ability to perform the basic duties of packet delivery and maintain an acceptable level of service for its users.

We observe that the vast majority of volumetric DoS attacks rely on one of two packet types: TCP SYNs (the first packet in the 3-way handshake used to establish a TCP connection), and short- or zero-payload UDP packets. These packet types share several commonalities which make them compelling to attackers:

- They are simple to generate. More specifically, it is easy to generate a *valid* SYN or UDP packet that a very large number of receivers will willingly accept. Each individual packet looks perfectly innocuous, but attackers can generate large quantities of them and require no special knowledge about their target(s) to do so.

- They cause harm immediately upon receipt. SYN packets prompt TCP servers to allocate state for a connection and generate a SYN-ACK in response. UDP packets prompt servers to check for running processes on the specified port – this is not an particularly complex process, but when receiving thousands or even millions of packets per second it causes significant delays.

- They are very short, which causes additional harm at every hop towards their destination. A SYN may be as short as 40 bytes (including the 20-byte IPv4 header) and with no payload an IPv4 UDP packet is just 28 bytes. Each packet incurs some fixed overhead to perform a routing table lookup, decrement the TTL field, and update the IP checksum. By sending many small packets rather than fewer larger ones, attackers congest forwarding infrastructure in the network and force traffic to be dropped before it even reaches the targeted destination.

Fortunately we observe an additional similarity which offers a means to address these vulnerabilities. Though legitimate clients do commonly send packets that look indistinguishable from these attack packets, they tend to do so at a significantly lower volume. A single attacker may send upwards of a hundred thousand packets per second (100 Kpps) – the threat of botnets comes not only from the large number of devices, but also the fact that each device acts greedily – particularly those able to spoof their source addresses. Indeed, as discussed in Section §6.4.1, we are able to generate SYN floods as large as 446 Kpps (164.1 Mbps) from a single device. By contrast, most clients likely average less than *one* SYN per second. Typical behavior is for clients to send a single SYN to establish a connection which is then kept open for some extended period of time to continue exchanging data. Initialization of new connections happens largely at the human-layer, for example when a user types a new URL into their browser or clicks on a link. More complex applications may initiate a burst of new TCP connections to pull data from multiple sources, but we should still expect many orders of magnitude fewer SYNs per second than an attacker would generate.

A similar disparity exists for UDP. While clients may legitimately send large quantities of UDP packets in general, they rarely have good reason to send packets with extremely short or non-existent payloads. As its name suggests the entire purpose of the User Datagram Protocol is for users to exchange data, and that data is carried in packets' payloads. There may be special cases in which short packets are used for signaling, based solely on the IP 4-tuple of source and destination addresses plus port numbers, but transiting such packets through the network is extremely inefficient. Again, we expect such packets to be sent many orders of magnitude more frequently by attackers than anyone else.

These natural asymmetries between clients and attackers offer an opportunity for a novel approach to volumetric DoS mitigation we call Trustworthy Independent Packets (TIPs). The essential concept is to make the generation of certain packet types marginally more difficult, in a way that significantly rate limits attackers while having a negligible impact on others. This allows us to bootstrap trust between unfamiliar devices – receivers are able to verify that senders expended some amount of resources to generate a packet, which implies that they must not be generating excessive quantities of traffic. Congested receivers can then prioritize packets that contain adequate TIPs over those that do not. Attackers must choose to expend resources generating TIPs at a reduced rate of attack, or else send packets without TIPs which can be easily filtered and dropped. In either case, the attack's volume is reduced. Our approach is also designed to permit verification *anywhere* in the network, not just at the destination, which allows packets to be dropped closer to their source, minimizing resource consumption at the receiver and at each hop along the way. Tuning the difficulty of TIP generation allows us to optimize the network's availability and performance by trading-off better protection from attack with lower overhead during normal operation.

This is a general concept that may be applied to many different attacks across different protocols, in slightly different ways. Anywhere there is a disparity between attacker and client behavior, we can leverage it to make the attacker's task more challenging. Specifics of implementation details may vary greatly across protocols however, and a careless implementation could easily be counterproductive. In the remainder of this chapter, we first present a broad overview of our TIPs approach and considerations we expect will apply to all implementations (§4.1). We then define two specific implementations of the concept in detail, both of which are designed to combat TCP SYN flooding attacks but in ways that are tailored towards different assumptions about attacker capabilities (§4.2). First we discuss eBPF, the tool used to implement our mitigations and our reasons for selecting it (§4.2.1). We address the threat of bandwidth-constrained flooders with SYN Padding (§4.2.2), which simply increases the minimum size of SYN packets, and SYN PoW, which adds small proofs-of-work to SYNs in order to rate-limit CPU-bound attackers(§4.2.3). We briefly summarize our mitigation designs in Section §4.3.

4.1 General Design

The core principle of our Trustworthy Independent Packets concept is for certain types of network packets to carry within them some entirely *self-contained* proof that the sender is relatively trustworthy, at least compared to some other senders. This trust is derived from an assumption that no device can possibly generate such proofs in dangerously large quantities.

If a sender includes a proof in their packet it means they must have been devoting significant resources to its generation, and therefore they must not have been devoting those same resources to generating a flood of traffic. This can also be thought of as packets "paying" (or "tipping") for the services they solicit. Not in the sense that any real or even virtual asset is changing hands from the packet's sender to its receiver, but that the sender has spent some limited resource to generate the packet as a show of good faith to the receiver. This allows us to redistribute the costs of DoS mitigation, by both making attacks more expensive to launch and dropping flood traffic closer to its source.

More concretely, a TIP-based mitigation involves two types of participants: **provers** and **verifiers**. Provers add some information to packets that proves they have consumed some excess resource, and verifiers decide how to handle received packets based on the presence or value of that proof. In the simplest case the packet's source host acts as the prover and the destination host acts as the verifier, but this may not always be the case. Performing verification closer to the source can allow earlier drops of malicious traffic, minimizing wasted resources in the network and at targeted servers. There may also be reason to offload the task of proof generation from clients to Network Address Translation (NAT) boxes or other proxy devices that are already familiar with and trust the client. For example, if a user owns legacy devices that cannot be updated they could deploy a prover on a separate proxy device in their home network, or their ISP could offer to provide a NAT box with a prover built in.

The precise form the proof takes will vary depending on the attack being mitigated and various properties of the network under attack, but generally it should require the consumption of some resource that is believed to be a limiting factor in the attack's generation. Some attackers may opt to send below their maximum possible rate to avoid detection, but many attempt to send as quickly as possible. Those are typically limited by either CPU or bandwidth – when sending large volumes of nearly identical packets we've found it's significantly fastest to modify a single buffer instead of loading pre-built packets from memory, so volumetric attacks are unlikely to be limited by attacker memory or disk storage. Ideally we should design both CPU- and bandwidth-based versions of TIPs for each vulnerable protocol, combining the two in a single packet design if possible.

For each of these attacker bottlenecks, there exists a general purpose tool we can use to increase the difficulty of flood generation. In the case of bandwidth-limited attackers, mitigation is as simple as increasing the minimum packet size. Larger packets serve as proof that senders have expended more bandwidth. If minimum packet lengths were doubled, the packet rate of bandwidth-bound attackers could be cut in half overnight. For CPU-bound attackers we turn to proof-of-work (PoW) systems. Today these are most commonly discussed in the context of crytpocurrencies and distributed consensus, but they were first developed for spam e-mail mitigation and function as general purpose tools for rate-limiting CPU-bound attackers, as discussed in Section §3.4. Requiring clients to encode a small PoW in each packet precludes attackers from generating large quantities of packets, because they lack the CPU resources necessary to compute proofs for each one.

This still leaves many questions about implementation details. Where in the packet should the padding/PoW be located? How much padding should be added, and how much work should be proved? Are these parameters fixed or variable? If they are variable, how are they communicated to endpoints? At what point in the packet generation pipeline should the proof/padding be added, and at what point in the receiving pipeline should it be verified? Where in the network should verifiers be located? Each of these questions must be addressed separately for each protocol or attack vector we aim to protect against. In the following section we tackle them in the context of TCP SYN floods.

4.2 Implementations in TCP

This section describes in depth the implementation of our two novel mitigations against TCP SYN Flooding attacks. We first discuss eBPF, the specific technology we use, including why we selected it and the common program structure we use for both mitigations (§4.2.1). We then expound on exactly how SYN Padding (§4.2.2) and SYN PoW (§4.2.3) operate.

4.2.1 eBPF

We have chosen to implement both our SYN Padding and SYN PoW mitigations as Extended Berkeley Packet Filter (eBPF) programs. eBPF enables us to deploy efficient, portable extensions to the Linux kernel without changing any kernel code or loading kernel modules [66]. This is ideal for experimentation as it enables rapid iteration. We can test, modify, and re-test different versions of a mitigation, with the process of re-compiling and swapping in a new version taking on the order of a second. It also facilitates incremental deployment since eBPF-based mitigations are lightweight and portable – compatible devices can install them without needing to perform a kernel or OS update. On the ingress side these programs operate before the main packet processing pipeline, allowing us to drop packets before they are able to consume significant resources. Indeed, eBPF is already in use by volumetric DoS mitigation service providers, who attest to it successfully dropping millions of packets per second.**test-L4drop** On some modern hardware, eBPF programs can be offloaded to the network interface card (NIC) to drop packets even faster and earlier. On the egress side it allows us to arbitrarily modify packets in-flight with full read/write access to the packet buffer. This lets us test mitigations that alter core Internet protocols in interesting ways, again without needing to touch a single line of the kernel code that implements the standard versions of those protocols. If these mitigations prove as valuable as we hope, we would expect to see them undergo further refinement and standardization culminating in a more direct in-kernel implementation, but for proof-of-concept mitigations at the transport and network layers eBPF is an extremely useful tool.

There are some notable drawbacks however. All eBPF programs must pass a strict verifier, which is how the code supplied at runtime from user-space is safe to run with kernel privileges. This places some restrictions on the programs we are able to write. First, they can only use a subset of the C programming language, excluding many common libraries that would ordinarily simplify routine tasks. Second, they must perform extremely diligent bounds checking when parsing packets in order to prevent illegal memory access. This does not really prohibit any categories of programs (except sloppy ones), but it does make the development process a more tedious. Third, programs must complete within some fixed number of instructions per packet. The exact limit is device specific – in older Linux versions it was set to just 4,096 instructions, but with the release of kernel version 5.2 in April 2019 it was raised to one million [67]. Determining if a program will halt is a famously undecidable problem in computer science, so enforcing this restriction requires that any loops must be unrolled by the compiler, which makes it challenging to do simple tasks like processing variable length fields (i.e. TCP options). Fortunately we are able to implement both of our mitigations within these constraints, and we believe the benefits to efficiency, portability, and flexibility of experimentation are well worth the trouble.

Each of our mitigations is implemented as a single eBPF program containing two sections, one that is attached to a device's network egress and one to its ingress. The egress section performs the prover role (adding padding or a PoW to outgoing packets) and the ingress section performs the verifier role (checking for proofs on incoming packets). Any eBPF-compatible device may opt to install one or both roles on any or all of its network interfaces. Wherever possible, the prover should be run directly by clients originating SYN packets, since it is their sending rate we are trying to limit. However, our mitigations are designed in such a way that a NAT box or other proxy device that has an existing trust relationship with a client may opt to add proofs to packets in transit, if for some reason the client is incapable of doing so itself. This may also be helpful for incremental deployment – many ISPs provide in-home NAT devices to their customers, which could be used to quickly roll out the prover side of a mitigation without requiring users to update their own devices. Verifiers can derive just as much trust from proofs inserted by such a proxy as from those generated by the original sender. They still attest to the fact that some additional bandwidth was consumed on the proxy's egress, and the preexisting trust we assume between the proxy and the original sender makes bandwidth consumption at the sender irrelevant. If a proxy decides to add proofs for large numbers of (potentially malicious) clients it places only its own resources at risk of exhaustion, and ISPs already have human-level strategies in place for identifying and penalizing their own misbehaving customers.

Additionally, TCP servers are not the only location where we may want to run verifier programs. Offloading them even a single hop to a firewall device can have massive performance benefits, and pushing them towards the network edge (closer to attack sources) is better still. Recall that our goal is to minimize costs to the targeted devices – removing the burden of verification from the server allows it to allocate all available resources to the task of serving clients. In-network devices are already making decisions about where and whether to drop or forward packets based on header data, and are therefore optimally positioned to serve as verifiers. Many of them are likely already running eBPF programs for other purposes, as it is a popular general purpose tool for network monitoring and administration – in those cases much of the overhead our verifier would incur is already being shouldered.

There is also benefit to running a large number of verifiers in multiple locations. Unless we can predict where attacks will originate, a widely distributed set of verifiers gives us the highest likelihood of dropping traffic close to its source. The economics are similar to those of ingress filtering, wherein everyone upstream benefits from dropping traffic as early as possible, but in our case verification can be performed 2, 3, or even 50 hops away from the attack source just as easily as it can by the sender's first-hop service provider. As we will demonstrate in Chapter §6 the overhead of verification is extremely small, meaning the benefits of detecting and dropping bad packets early far outweigh the risks of *re*-verifying good packets at multiple locations. If overhead does prove burdensome, verifiers could choose to perform their task on only a random sampling of the traffic they process. With a verifier at every hop, they could collaborate to ensure most traffic is verified *somewhere*, despite only a small portion being verified *everywhere*.

4.2.2 SYN Padding

Our first mitigation, SYN Padding, is an extremely simple concept. Essentially all we are proposing is to increase the minimum length of TCP SYN packets. This is intended as a mitigation against SYN flooders that are bandwidth-constrained, by forcing them to send fewer, larger packets. The primary target of modern high-volume attacks is the network itself – forwarding overhead is predominantly per-packet rather than per-bit, so large quantities of small packets are the most effective way to overwhelm routers and force them to delay or drop legitimate traffic. By most accounts the vast majority of real-world volumetric DoS attacks rely on either TCP SYN packets, which may be as short as 40 bytes (including the 20 byte IPv4 header) or zero-payload UDP packets, which are just 28 bytes including the IPv4 header.

While there may be benefits to padding UDP packets to some degree as well, we choose to focus on SYNs in this initial proof-of-concept for two reasons. First, they are more situational, being generated relatively rarely by legitimate clients at the beginning of potentially long-running communication sessions. This quality is easily discernible from standard flags set in every TCP packet header, enabling efficient filtering. UDP packets are much more generic by comparison. If one is being used to initiate communication, as in a DNS request for example, that fact can typically only be learned by reading application-layer data. The UDP header itself only tells us what type of application is being used, not *how* it's being used. Extracting and acting on this higher-layer information poses more technical and ethical concerns than special handling for TCP SYNs, as discussed in Section §3.2.1.

As discussed at the beginning of this chapter though, there is little reason for legitimate clients to send extremely short UDP packets. In reality this is not actually a flaw with TCP or UDP, but rather a vulnerability born of inefficiency in IP itself. Attackers can exploit the short minimum packet length to exhaust forwarding resources regardless of what transport-layer protocol is used. The precarious relationship between packet size and network efficiency has been known since at least as early as 1976, when Metcalfe and Boggs published the original Ethernet paper and observed significant drops in efficiency with short packets [68]. The Internet architecture has changed a great deal since then but the same fundamental problem persists. It is rarely discussed in the context of DoS mitigation, but we assert that the problem of volumetric DoS attacks is inextricable from those of network efficiency and fair resource sharing.

The second reason for starting with TCP is that it already includes a convenient standards-compliant mechanism for padding SYN packets. The minimum length of a TCP SYN packet is 40 bytes, 20 for the IPv4 header and 20 for the TCP header, but the TCP header can include up to 40 additional bytes of various options. Table 4.1 lists the three standard option types that all implementations must support, as defined in the recently released RFC 9293 [69]: End of Option List (EOL), Maximum Segment Size (MSS), and No-Operation (No-Op). Each TCP packet can only contain one each of the EOL and MSS options, so using those would limit us to at most 5 bytes of padding. Moreover, EOL will naturally be included in any packet that uses other option types, and many clients already use the MSS option by default. That leaves the No-Op option, which turns out to work perfectly for our purposes.

Ordinarily No-Ops are used to align other options at 32-bit boundaries for more efficient packet processing. They are simply a single byte with a value of 1 that tells the receiver to do nothing and continue on to the next byte. Being a single byte allows us tremendous

Code	Option	Length
0	End of Option List	1 Byte
1	No-Operation	1 Byte
2	Maximum Segment Size	4 Bytes

Table 4.1: The mandatory options that all TCP implementations must support, and the length in bytes each one adds to the TCP header. Additional options to support selective acknowledgement (SACK), timestamps, and window scaling are recommended, but only the three in this table are required for basic interoperability. Our SYN Padding implementation relies on an early End of Option List option, but copies of the No-Operation option can also be used.

granularity in parameterizing this mitigation, as we can add any number of No-Ops to each SYN from 0 to 40, at most doubling the packet's length. However, we recommend the maximum 40-byte padding as even that amount poses negligible overhead to ordinary clients, while maximizing the potential reduction in attack volume. This padding can be thought of as verifiable proof that a packet's sender has expended some amount of bandwidth resources. [1]

The following subsections detail how our eBPF prover adds this No-Op padding to outgoing TCP SYNs and how our verifier checks incoming ones. Figure 4.1 depicts both halves of the mitigation. The source code for our implementation of SYN Padding is provided along with the rest of our research data [3].

[1] These repeated No-Ops are a perfectly viable way to implement SYN Padding, but it turns out the EOL option actually can be used to the same effect. The TCP standard permits an EOL to be placed early, before the end of the options field, as long as the remainder of the header is all zero bits. This means that instead of adding No-Ops, we can simply update the data offset field to indicate an increase in the packet length, and fill the added bits with zeroes (the EOL option itself is also just a zero bit). Since newly allocated memory is regularly initialized to zero already, this approach should be marginally more efficient than setting the bits to ones for No-Ops.

4.2.2.1 Prover

The prover program runs anytime a packet is leaving the egress side of a link. It first checks the ethernet frame's protocol field to verify that it's an IPv4 packet[2], then checks the IPv4 header's protocol field to verify that it's a TCP packet. If either check fails, we send the packet on unmodified. For any TCP packets encountered, we next check that their SYN flag is set but their ACK, FIN, RST, and PSH flags are not – this distinguishes a "SYN" packet from other TCP control packets. Again, any other types of TCP packet are sent as-is. For SYN packets we check the IP header's total length field and compare it against our threshold of 80 bytes. Any SYNs that are already 80 bytes are sent unchanged. For shorter SYNs we compute the difference between the current length and the threshold, and expand the buffer by that amount. We set each of these newly added bytes to a value of 1, the option code for No-Ops, then update the Total Length field in the IP header and the Data Offset field in the TCP header. Finally, we update the TCP and IP checksums to account for the modifications and send the padded packet.

4.2.2.2 Verifier

The verifier begins in the same way as the padder: it first checks to see if an incoming packet is IPv4, then TCP, then a SYN. If any of those three checks fails it passes the packet on to the normal receiving pipeline. For SYNs, the IP header's total length field is checked and compared against the threshold of 80 bytes. Sufficiently long SYNs are received normally,

[2]We intend to develop and test IPv6-compatible implementations in the future, but also note that the 20-byte increase in fixed header size from IPv4 to IPv6 is already half the number of bits we are proposing to add as padding. This may mean that IPv6 is inherently more resilient to volumetric DoS attacks than IPv4 – we discuss this further in Chapter §7, but confirming this theory is left for future work.

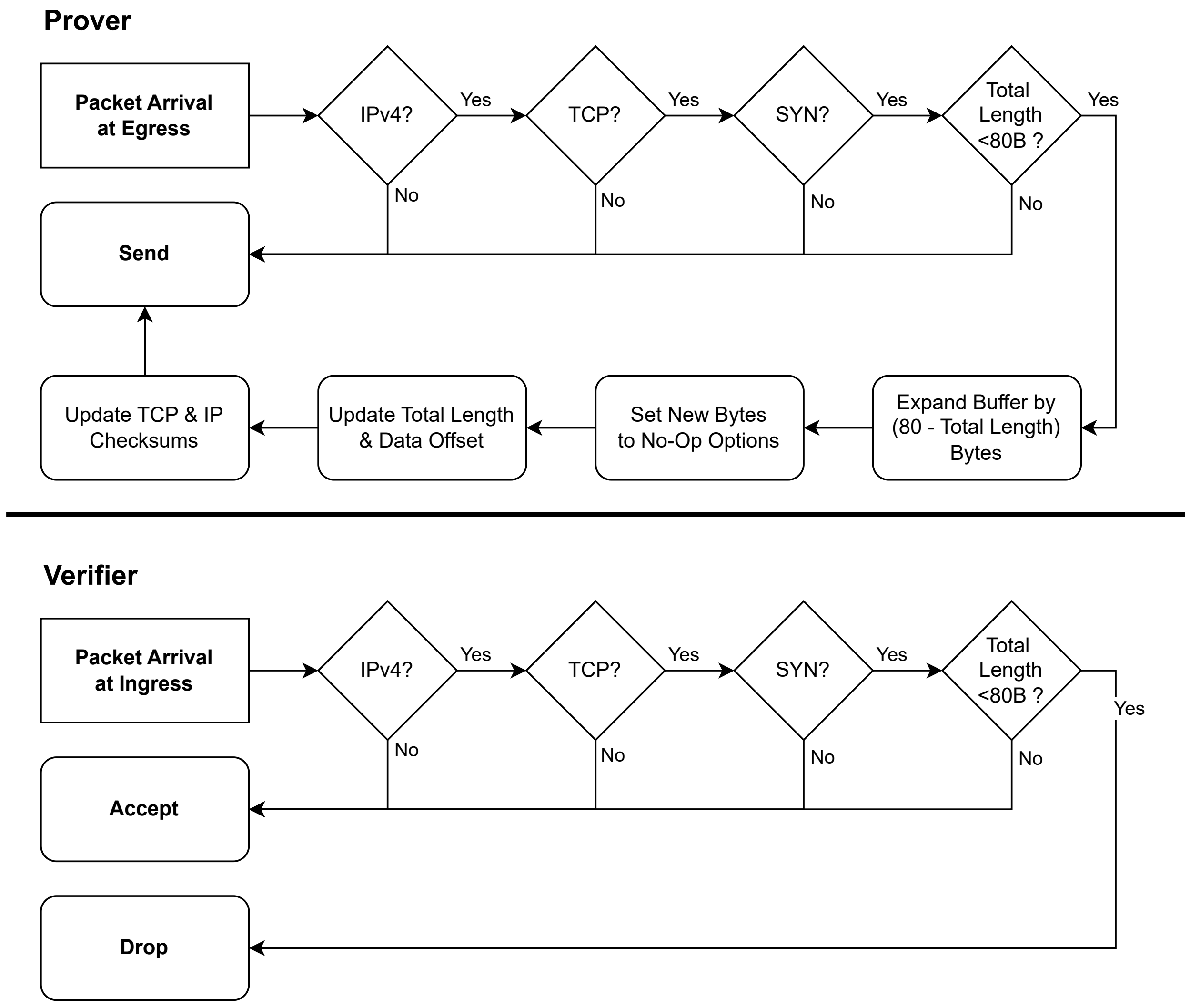

Figure 4.1: Basic operation of the prover and verifier roles in our SYN Padding mitigation. The prover pads all SYNs to a total length of 80Bytes before sending, and the verifier drops any incoming SYNs below that length.

while shorter SYNs are dropped.

Our expectation is that performing these checks in eBPF will be more efficient than actually receiving and continuing to forward extremely small SYNs. The closer to an attack's source the verifier can be implemented, the greater effect this mitigation will have, as it prevents every subsequent hop towards the destination from needing to expend forwarding resources.

4.2.3 SYN Proof-of-Work

Our second TIP implementation in TCP, SYN PoW, uses miniature proofs-of-work to combat high-volume TCP SYN floods launched from CPU-constrained devices. As discussed in Chapter §3, these are common characteristics of the devices from which DoS attacks are launched. Though modern DoS attacks tend to be widely distributed, each individual device in the botnets that launch them will often attempt to spoof its source address and to flood the network with as much traffic as possible. We note that there is an inherent asymmetry between SYN flooders and typically TCP clients – the quantities of SYN packets they send are multiple orders of magnitude apart. By making it just slightly harder to generate each individual SYN, we can leverage this asymmetry to significantly rate-limit attackers while having a negligible impact on legitimate clients.

The core concept is for TCP clients to encode a miniature proof-of-work (see Section §3.4 for a general overview on PoW systems) in each SYN packet, which can be efficiently verified by the receiver or an intermediary device. During times of congestion or under threat of attack, verifiers may choose to drop packets carrying lower-value proofs. For regular clients these proofs take only a small fraction of a second to generate, and are generated rarely. For attackers that may normally send hundreds of thousands of packets per second, those small fractions of seconds add up. They are left with two choices: either expend CPU resources to generate valid proofs for every SYN they generate, which requires extra time and slows down sending rates; or send packets with random-value proofs, the majority of which will be dropped by verifiers before they can cause significant harm. Either way the result is less junk traffic in the network.

As with SYN Padding we implement SYN PoW as ingress and egress sections of a single eBPF program, wherein the egress section performs the role of the **prover** and the ingress section acts as the **verifier**. Figure 4.2 illustrates the basic operation for each of these roles. As discussed above we expect the prover to be the packet's original sender, but our implementation allows it to be offloaded to a NAT or other proxy device if necessary. Similarly, one or more verifiers may be located anywhere in the network, ideally in multiple locations.

This mitigation is significantly more complex than SYN Padding, and there are a number of complicating factors in implementing it. First is that we need somewhere to encode the **nonce**, a portion of the message that can be randomized or incremented with each iteration of the prover's operation.[3] We also need to select a **hash function**, and determine what fields from the packet should be included as input to it. We refer to this input as the *digest*. Finally, we need to determine an appropriate proof **threshold**, in order to set the expected amount of work that must be done to generate each SYN. The following subsections address each of these complexities.

4.2.3.1 Nonce Location

The nonce is a portion of the digest that is randomized or incremented with each iteration of the proof-of-work protocol. Changing its value changes the output of the hash function in an unpredictable way. Repeatedly changing the nonce and recomputing the hash is the computational "work" our clients prove. Their objective is to find a nonce that, when included in the digest for the packet they want to send, produces a hash output above a certain

[3]Note that if designing a scheme from scratch we could allocate a dedicated field to this purpose.

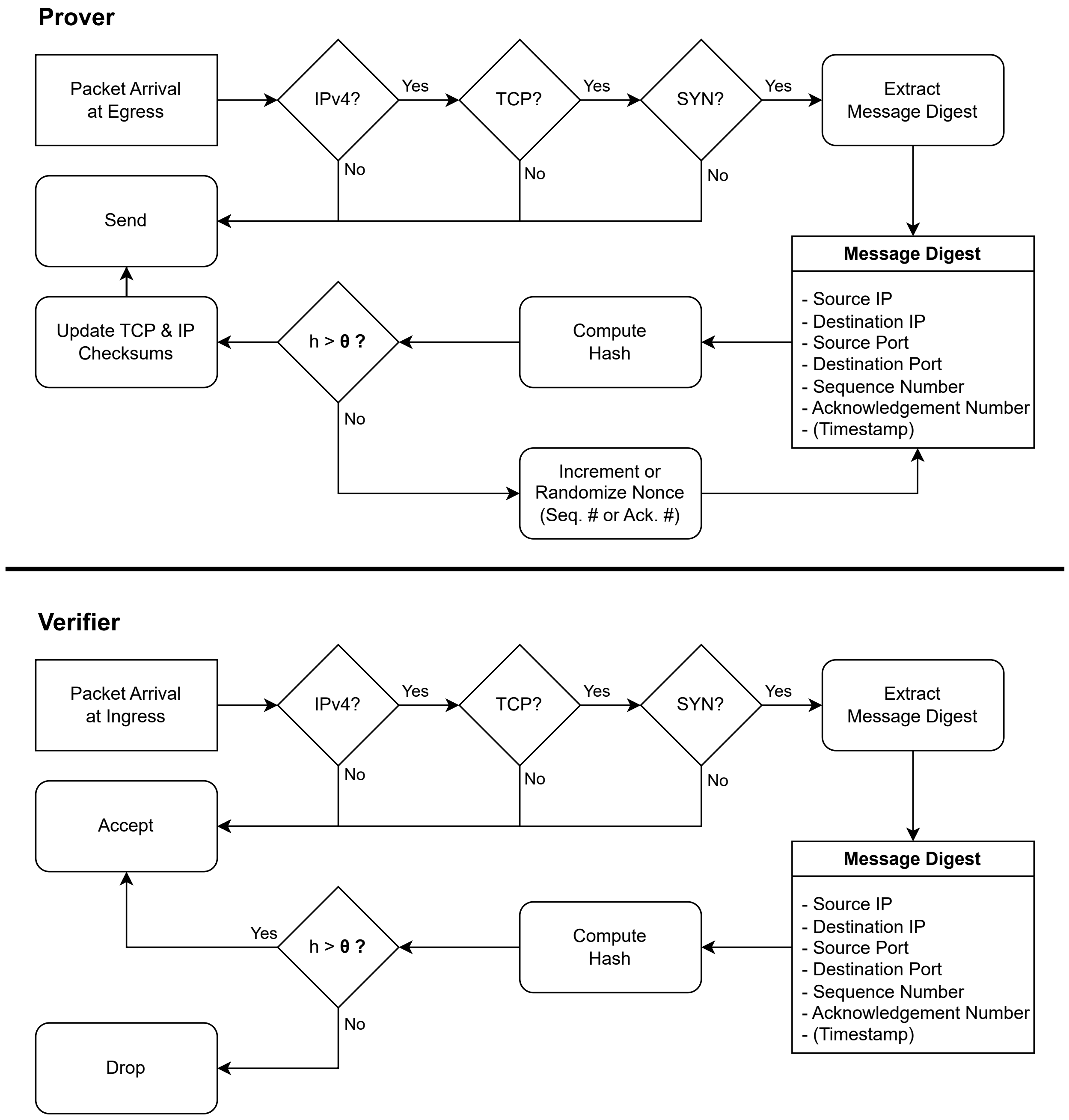

Figure 4.2: Basic operation of the prover and verifier roles for our SYN PoW mitigation, including essential contents of the message digest. Before sending a SYN, the prover repeatedly alters part of the message digest (the nonce) and recomputes the digest's hash value (h) until it exceeds some threshold (θ). The verifier then checks the hash values of incoming SYNs against its own threshold to decide whether they should be dropped or accepted. All other packet types are sent/accepted normally by both roles.

threshold, θ. Therefore the nonce needs to be some bits in the SYN packet that can be set to an arbitrary value by the sender. We must also consider the nonce's length, as it determines the granularity with which we can tune the mitigation. With a b-bit nonce each SYN can be made to have as many as 2^b different hash outputs (all other fields being unchanged), which gives us 2^b possible values for θ.

There are a few potential locations where a nonce could be added to SYN packets, with some natural trade-offs between them. One approach would be to define a new TCP option type specifically for this purpose, but our goal is to design an implementation that is as transparent and backwards-compatible as possible. Additionally, the unordered and variable-length nature of TCP options makes them extremely difficult to parse under the constraints of the eBPF verifier. Instead we want some way to encode the nonce in an existing field of the TCP header. Proof-bearing packets should look and act like regular SYNs. We have identified two potential header fields capable of meeting our needs: the Sequence Number and the Acknowledgement Number.

The Sequence Number field presents perhaps the most natural location for the nonce, but can only be used when the prover is also the direct originator of the SYN. Recall that the nonce is simply some value that can be randomized or incremented with each iteration of the proof-of-work protocol. It has long been recommended to randomize the initial sequence number (ISN) used in SYN packets to prevent connection hijacking attacks. A secure algorithm for doing so, defined in RFC 6528 [70], was recently included in the main TCP specification via its update in RFC 9293 [69]. Our SYN PoW prover would first use this standard mechanism to generate an ISN, then increment it with each hash iteration. This change makes no difference to the receiver or in-network devices, as they have no expectation for what the ISN should be, and we see no reason it would diminish the security of the random generation mechanism.

The one device that does care if this number changes is the one that generates it, which is why proxy-based provers cannot use this implementation. If a proxy were to change the ISN in flight, the original sender would not recognize the SYN-ACK it received in response, preventing connection establishment. The same problem prevents us from using eBPF to implement a prover locally on the SYN's originator, because our egress eBPF program acts on packets *after* the ISN has been set in the kernel. It may be possible to circumvent this by having our prover update the socket metadata that tracks the ISN, but this would add unnecessary complexity and overhead. We could also implement a two-way proxy that translated sequence/acknowledgment numbers back and forth between their pre- and post-proof values, but that too would add considerable complexity and require ongoing packet modification and data storage throughout each TCP connection – our goal is to modify and verify *only* the initial SYN packets.

Instead, our eBPF prover encodes its nonce in the Acknowledgement Number field, which is currently unused in SYN packets. It is typically initialized to zero but neither the original nor recently revised TCP standards specify any initial value for it [69], [71]. We have found that both local eBPF provers and proxy provers are able to modify this field arbitrarily without any impact on a client's ability to establish a connection. The Acknowledgement Number field therefore provides an ideal location to encode a nonce: 32 bits of free, unused header space, which is extraordinarily rare to come by in a protocol as lean as TCP.

We do note that SYNs with non-zero Acknowledgement Numbers (NZAN) elicit a warning (though not an error) from Wireshark when viewing packet captures. There is also a chance that some middleboxes might zero-out the field in-flight despite non-zero values being fully standards-compliant, but we have yet to see any evidence of this. Neither our testbed hardware nor our client-server applications had any issues forwarding or processing NZAN SYNs. However, to ensure maximal compatibility we would recommend eventual in-kernel implementations use the Sequence Number field, while proxies and eBPF implementations use the Acknowledgment Number.

It is possible for both versions to coexist side-by-side, simply by including both fields in the digest. That way verifiers perform exactly the same operation regardless of where the prover was located or which of the two fields it used. This allows us to use eBPF and proxy implementations for initial tests and deployments, before gradually transitioning towards more in-kernel implementations as the mitigation is refined and standardized.

4.2.3.2 Digest Composition

Next we consider what information from the SYN should be included in the packet's **digest**, what is input to the hash function for the proof-of-work protocol. Again, we include both the Sequence Number and Acknowledgement Number fields as potential nonce locations. Local provers may use either one, while proxy provers and eBPF implementations must use the Acknowledgement Number. We must also include some information that the attackers will change with each packet, to ensure they cannot simply find a single high-value nonce and reuse it with every packet. For address-spoofing attackers, this can be guaranteed by including the source IP address. To account for non-spoofing attackers we include the entire 4-tuple of source address, source port, destination address, and destination port. If an attacker were to keep these four fields unchanged with every packet they would quickly succeed in creating a half-open connection at the server, after which point their subsequent SYNs would be ignored as duplicates. Completely identical packets would still consume forwarding resources in the network, but there are various techniques for efficient "packet deduplication" to protect against such floods [72]–[75]. We emphasize that our mitigations are designed primarily to supplement such strategies, not necessarily to replace them.

In theory attackers could pre-generate large quantities of SYNs with different addresses and ports that all have valid proofs and send them from memory, but in practice we expect that would result in significantly slower floods from most devices. This will depend on resource bottlenecks of individual attackers, but in our own attempts at generating such high volume spoofing attacks we've found the fastest method is to allocate a single buffer, fill it with the data for the first packet, and overwrite specific bits in place as necessary for subsequent packets. Any operations that copy data to new buffers or read whole packets in from memory will slow this process down significantly, well beyond the cost of recomputing TCP and IP checksums for each packet and beyond the cost of our miniature proofs-of-work as well.

Attackers also have the option to *not* attempt the proofs and continue sending "dumb" floods. In this case their nonce will be whatever default value is selected in normal packet generation – the normal ISN for the Sequence Number field, and zero for the Acknowledgement Number field. Verifiers can still compute hashes for these dumb packets in exactly the same way they do for "smart" ones. Attackers will occasionally get lucky and stumble upon a packet that has a high hash value with these default fields, but the vast majority of dumb packets will be dropped as soon as they reach a verifier. Still, verifying and dropping these packets does require some resources, which means dumb floods may still be viable. We evaluate the impact of both dumb and smart floods under SYN PoW in §6.4.

If further protection against replay attacks is desired, a timestamp can also be included in the digest. Verifiers would drop any packets with out-of-date timestamps before even needing to compute their hash, forcing attackers to include a recent timestamp in each packet they send, which in turn would force them to compute unique proofs. We could incorporate the standard TCP timestamp option, but it is not supported by all devices and as mentioned above TCP options are difficult to parse in eBPF. Another possibility is to sacrifice some bits from the nonce to encode our own coarse-grained timestamp there. For example, we could use the first 16 bits of the Sequence/Acknowledgement Number field to store the nonce and the second 16 to store a local timestamp. With a granularity of one second-per-bit this would provide just over 18 hours worth of unique timestamps, which is clearly insufficient. This approach would also require verifiers to know which of the two fields the prover used, and for provers to modify such a large portion of the Sequence Number field in such a predictable way would likely compromise the security of the ISN generation function defined in RFC 6528 [70]. Additionally, any mechanism that uses timestamps must rely on clock synchronization among devices, which is absent in many real-world deployments. The Network Time Protocol (NTP) on which clock synchronization relies [76] also adds more communication to the network when our goal is to reduce congestion.

Future refinement of this idea may prove that some form of timestamp is in fact necessary, but for our initial experiments we assume replay attacks are not a viable threat for high-volume spoofers under this model. Our digest contents are therefore the two potential nonce fields (Sequence Number and Acknowledgement Number) and the full source/destination address/port 4-tuple. These digest contents are listed in Table 4.2, along with their corresponding data lengths.

Field	Length
IPv4 Source Address	4 Bytes
IPv4 Destination Address	4 Bytes
TCP Source Port	2 Bytes
TCP Destination Port	2 Bytes
TCP Sequence Number	4 Bytes
TCP Acknowledgement Number	4 Bytes

Table 4.2: Contents of the SYN PoW digest that is input to the hash function. Source and destination ports are included to protect against spoofing attacks, while the sequence and acknowledgement number fields are used to encode the nonce.

4.2.3.3 Hash Function Selection

Generally, a hash function is some mathematical operation that takes either a fixed or arbitrary-length input and produces a fixed length output, in such a way that guarantees two key properties:

1. They are **deterministic**: for any given input the function always returns the same output.

2. They are **one-way**: it is provably difficult to predict the output any given input will produce, or to find an input that will produce any given output.

The first property ensures that all participants agree on a packet's **hash value**, the result of hashing its digest. As long as you know what hash function to use, each packet's value is self-evident. The second property prevents attackers from gaining an unfair advantage. The exact definition of "provably difficult" depends on the specific mathematical formulation used, but the general concept is that there should be no method of finding an input that yields a given output that is more efficient than random guessing. This means the best way for provers to find a nonce that yields a sufficient hash value is by trial-and-error, brute-forcing the hash function with random nonces, thereby performing the work their packet hash serves to prove.

For our purposes there is an additional constraint: we need a hash function that is simple enough to be implemented in the subset of the C language that is available to eBPF programs, and it must be able to pass the eBPF verifier. Many standard libraries are unavailable which precludes the use of most common cipher suites, and restrictions on the instruction count, loop bounds, and memory access prevent us from re-implementing those tools. A potential future in-kernel implementation would not be subject to these limitations, and could make use of a common hash standard such as SHA-256 or MD5. As long as all participants agree on the same function, it makes little difference to our implementation what that function actually is – future revisions may swap in something simpler, more efficient, or more secure than what we have chosen.

We still prefer to use an off-the-shelf implementation for this component of the system rather than trying to develop a new custom hash function. That is a notoriously difficult task to get right, and better left to those with more expertise in the field of cryptography. Based on the criteria above, we have chosen to use the "SuperFastHash" developed by Paul Hsieh [77] for our initial implementation. Hsieh's algorithm is less than 50 lines of C code, and only requires inclusion of the `stdint.h` header. We are able to run this code, unmodified, in eBPF without issue. Another benefit is that, as the name suggests, each iteration of the SuperFastHash can be computed quite quickly. While it may seem counter-inutitive, this is actually a desirable property for our application because it gives us more control in tuning the system. Our strategy is to make attackers complete a large number of simple proofs rather than a single challenging one. Making it easy to generate a single proof allows us to minimize overhead at legitimate clients, while attackers are still stymied by the sheer quantity of proofs they must generate. Regardless of the hash function chosen, we are limited to a relatively low maximum per-packet delay because we cannot exceed the one million instructions-per-packet limit set by eBPF. For reasons discussed in the following section we do not expect this limit will pose an issue. Hsieh also notes that his SuperFastHash has reportedly been used in such notable software as the Apple Safari and Google Chrome browsers, which indicates that it has likely undergone extensive vetting for security properties, beyond the already extensive analysis provided by Hsieh himself.

4.2.3.4 Setting Proof Thresholds

Once a hash function is selected, the next question is how to set the proof-of-work threhold θ. Our goal is find a value that requires provers to perform an average of k hash iterations per packet in order to find a nonce that yields a hash value $h \geq \theta$. The relationship between k, θ, and the hash output length b is defined by Equation 4.1.

$$\theta = 2^b * \frac{k-1}{k} \tag{4.1}$$

A hash function with a b-bit output allows for 2^b possible values of h. If we want clients to take k attempts on average to generate a valid SYN, then only $\frac{1}{k}$ randomly generated SYNs should be valid, and therefore any SYN with $h < 2^b * \frac{k-1}{k}$ should be dropped.

The next question then is how much work we actually want clients to prove. We know how to set θ to target a desired value of k, but how do we know which value of k is optimal?

Very small values of k will have little effect on attackers, while very large values will have unacceptable overhead for regular clients. We want to learn where the sweet spot is in between those two extremes, and how that may depend on the nature of the attack, the type of client application, the topology of the network, etc..

If we assume that every device in the system takes t seconds to perform a single iteration of our hash function, then an iteration target of k adds a delay of $t * k$ seconds to the sending of each packet, plus an additional t seconds for the single hash iteration performed at the verifier. That same target also limits attackers to at most $\frac{1}{t*k}$ packets per second. Since attackers don't wait for SYN-ACK responses, the additional verification delay does not reduce their sending rate further. For a concrete example, let's say $t = 1ms$ and $k = 10$. That would add 11ms of latency to the initialization of each TCP connection for legitimate clients, but in return attackers would be rate-limited to at most 100 valid SYN packets/second. These are encouraging numbers – 11ms is not an unreasonable amount of time to make clients wait for such a rarely sent and potentially damaging packet, and 100 packets per second is many orders of magnitude less that what attackers are typically capable of generating. There appears to be ample room to tune this system to prioritize either stronger rate-limiting or lower overhead: depending on context we might prefer a 1.1ms delay with a 1000 packet/second rate-limit, or a 110ms delay with a 10 packet/second limit.

In reality though t is not constant across devices, which means the optimal value of k can only truly be determined through experimentation in-context. Generally we expect these differences in hashing efficiency to favor defenders over attackers. As discussed in Chapter §2, a large portion of attackers are resource-constrained IoT devices. We expect them to have under-powered CPUs, and to therefore exhibit higher than average values of t, allowing us to rate limit them even more severely. Conversely, we expect verifiers would primarily be deployed on the sort of high-powered enterprise network devices that are already typical in volumetric DoS mitigation. These may be able to achieve below-average hash rates, slightly decreasing the latency experienced by legitimate clients. Clients themselves run on diverse hardware – averaging across all clients we would (almost by definition) expect to see average values of t.

4.2.3.5 Communicating Proof Thresholds

Finally, there is the issue of communicating the proof threshold value θ. To maximize efficiency, provers should know what hash values will be accepted. That allows legitimate clients to *only* send packets that they know carry a sufficient proof, and to therefore avoid wasted work and unnecessary retransmissions. This is challenging to achieve without violating the core principle of TIPs though, that we need to build trust in the *very first* communication between two devices. If the SYN is the first packet exchanged between a client and server, then the client must somehow know how much work the server and/or other verifiers on the path to the server are expecting it to prove. There are many possible ways this might be achieved, but we have identified three plausible solutions: setting a standard global threshold, communicating thresholds indirectly through DNS, and inferring thresholds based on packet loss.

4.2.3.5.1 Global Standard The simplest solution would be to establish a common global threshold across the entire Internet. All provers would be pre-configured to perform the same amount of work, and all verifiers would reject SYNs with hashes below the same value. There are very good reasons to value the simplicity of this approach, particularly the complete lack of communication it requires. While this may indeed prove to be the best overall solution, it cannot possibly provide optimal mitigation efficacy in every context. As mentioned above, the ideal proof threshold depends on parameters of the system being measured, including but not limited to the amount of time it takes each device to complete a single hash iteration and how many packets each device ordinarily attempts to send per second. While there may be some θ value that yields reasonably high efficacy and low overhead across a variety of realistic scenarios, we first consider methods by which thresholds could be communicated *indirectly.*

4.2.3.5.2 DNS One compelling option is for servers to announce their θ values via the Domain Name System (DNS). The average Internet user does not actually know the IP addresses of servers with which they want to establish a connection (most probably don't even know what an IP address is). Instead their device first queries a DNS server using a more memorable, human-readable domain name, and that server responds with one or more corresponding addresses. This too is becoming an increasingly outdated form of interaction with the Internet. Users rarely type or even look at domain names, instead typing search terms directly into their web browser and following links served by a default search engine, or else using mobile/desktop applications that abstract and obscure device identities and interactions even further. DNS is still used heavily in these cases though, as the IP addresses associated with the search engine, link destinations, and back-end application servers are all subject to change over time.

There is a strong precedent for such use of the DNS to deploy security extensions on the Internet. The most notable example is DomainKeys Identified Mail (DKIM), which uses the DNS to store public keys for the purpose of signing and encrypting e-mail [78]. Adding large amounts of extraneous data to DNS entries has the negative side effect of increasing the amplification factor that can be attained in DNS reflection-based DoS attacks; attackers send short queries spoofing the source address of a targeted host, prompting DNS servers to flood that target with large unwanted response packets. DKIM requires signers to use 1024-bit or longer RSA keys, while the threshold information our SYN-PoW implementation would add requires only 32 bits.

The results of DNS queries are often cached for future use so clients may not send one before every SYN, but most will at least perform a DNS lookup before the very *first* SYN sent to a new server. Cache entries are set to expire to avoid stale information though, in some cases after as little as 30 seconds or as long as a month, but typically within 24 hours – after which point if the client wants to reconnect to a "known" server it will first send a new DNS query before sending a new SYN. This means that if servers communicate their proof-of-work thresholds through the DNS, we can expect clients to receive up-to-date information within roughly one day. Such infrequent updates could accommodate hardware upgrades or other major changes to the network, but are clearly insufficient for real-time mitigation. In 2022 Microsoft Azure reported [9] that 89% of all observed DoS attacks lasted less than an hour, and 26% lasted only 1-2 minutes.

Slow propagation of threshold information is not the only hurdle to implementing a DNS-based mechanism. It involves some prior communication before the SYN is sent, which we explicitly aim to avoid. That communication is not directly between the prover and verifier, nor between the client and server, so it does not directly worsen the situation we are trying to improve. This would however create a new incentive to attack DNS servers, in much the same way that traditional client puzzles often see their puzzle generation and distribution mechanisms become attack targets.

Relying on DNS would also significantly complicate the distributed nature of our mitigation. We want verifiers to be located in the network instead of just at the server, which means the DNS server would need to return the *maximum* threshold across all verifiers on the path from the source to the destination. It's not clear how that information could be collected, and if it could there would be significant overhead involved with both gathering and storing it. Provers dislocated from clients would need to perform their own DNS lookup for each SYN before adding a proof and forwarding it along. Unless a proxy prover already has this DNS information cached locally this lookup will add significant latency, and even a local cache cannot easily be accessed from within the confines of an eBPF program.

4.2.3.5.3 Inference It may also be possible for provers to infer proof thresholds on their path over time, through an extension of this concept we call **Enforceable Congestion Control** (ECC). If a client sends a SYN and does not receive a SYN-ACK, they can infer that the packet was dropped due to congestion at the server or in the network. Just as traditional TCP congestion control interprets dropped data packets as a signal to halve the window size [79], ECC takes dropped SYNs as a signal to double the prover's k value which has a similar effect of halving the rate of traffic generation. On receipt of an ACK, ECC decrements k which linearly increases the sending rate.[4] As with TCP congestion control,

[4]In practice, devices will be adjusting their θ values, as k is an expected average based on θ. However, it is the expected number of hash iterations that we aim to double and decrement with ECC, so we find it more intuitive to discuss changes in terms of k.

there are likely more efficient algorithms than this simple Additive Increase Multiplicative Decrease (AIMD) approach, and our provers would be similarly free to select whichever they prefer.

Verifiers would then be free to adjust their thresholds in response to changes in demand, without actually needing to announce the new value. This requires proxy provers to be on both the outgoing and return paths in order to keep track of SYN/SYN-ACK pairings. We assume such proxies will be located close to the clients they add proofs for, often on NAT boxes one or two hops away, or even virtual devices running on the same hardware as the client. In these cases the prover will naturally see every SYN and SYN-ACK between the client and server, allowing it to adjust its k value accordingly.

This approach could also make things more difficult for address-spoofing attackers, because they have no mechanism for detecting when their packets are dropped. Flooders don't expect responses to anything they send. This means they would have no signal from which to infer the threshold. Attackers that are naturally incentivized not to attempt the proofs would be unaffected, but "smart" attackers would struggle to find the level of proof effort capable of maximizing their potential for harm. Overestimating the threshold means rate-limiting themselves unnecessarily, and underestimating it means their packets will be dropped. We would expect to see legitimate provers' k values oscillate around the actual threshold similarly to how the iconic TCP sawtooth fluctuates around the ideal window size.

The major impediment to using ECC for threshold inference with our current SYN PoW implementationis that it adds significant complexity. Our current eBPF programs are entirely stateless, performing exactly the same actions on each packet regardless of the traffic that preceded it. For provers to learn thresholds over time they would need to maintain and reference some table listing expected thresholds for each destination address. With in-network verifiers it is reasonable to expect this table could be condensed to store only one threshold per /24 or even larger subnets, since the paths from a single source to every host in a given subnet are likely to transit many of the same hosts and in our case the same set of verifiers. For verifiers to adapt their thresholds based on current demand also adds some complexity, but checking the current status of system resources is largely what eBPF is designed to do.

4.3 Summary

To summarize, we are proposing a protocol agnostic approach to volumetric DoS mitigation based on increasing the difficulty of generating certain packet types that are sent disproportionately more often by attackers than others. Our **Trustworthy Independent Packets** (TIPs) can easily be generated by legitimate senders and verified by receivers. Attackers are left with a choice between expending additional resources to craft valid packets and slowing down their floods in the process, or sending mostly invalid packets that will be dropped before they reach their destinations. We observe that high-volume flooders are likely to be rate-limited by either their CPU or their bandwidth, which means we should develop mitigations in which the increased difficulty of packet generation involves consumption of those resources.

To that end, we presented two novel mitigations against the ubiquitous threat of TCP SYN floods. SYN Padding simply increases the minimum packet length, which forces bandwidth-constrained attackers to send fewer, longer packets per second, reducing per-packet forwarding overhead in the network. With SYN PoW, clients generate small proofs-of-work for each SYN to demonstrate that they have expended some CPU resources. In Chapter §6, we will evaluate both the efficacy and overhead of these proposed mitigations through controlled testbed experimentation, after discussing *how* to accurately evaluate them in Chapter §5.

Chapter 5

Metrics and Measurement Techniques

To facilitate accurate evaluation of the mitigation strategies presented in Chapter 4, we first establish a general framework for measuring the costs and benefits of DoS mitigations. Our framework builds on prior work evaluating the impact of DoS attacks, but extending impact metrics to the realm of mitigation is surprisingly non-trivial. We must consider a mitigation's efficacy at reducing the damage caused by an attack as well as its overhead outside periods of attack, both of which may be heavily influenced by: the type and volume of attack traffic; the type and volume of legitimate client traffic; the hardware and software used by clients, servers, and in-network devices; the latency, bandwidth, and loss rate of network links; the overall layout of the network topology and relative positioning of devices; and myriad other factors. Clients that normally communicate infrequently are less likely to notice a disruption; servers with fewer resources to start with are bound to reach their breaking point more quickly once those resources start being consumed; and clients directly adjacent to a server tend to experience less impact from attacks than those who are several hops away. Even the act of measurement itself will alter the results, so we must carefully consider what sort of measurements to take, where, and with what granularity.

We cannot hope to state whether any mitigation is objectively good or bad – rather, our goal is to determine which mitigation is best-suited for a given scenario, and in what sort of scenarios a given mitigation is likely to have the greatest effect. Our immediate motivation is to evaluate our TIPs in comparison with existing mitigations, and to compare different versions of our mitigations against one another in order to find optimal parameters. The framework this chapter presents has much broader utility however, as our definition of a mitigation is deliberately vague. These methods can be applied to evaluate essentially *any* modification to a network system. We envision this same framework being used to compare designs and implementations of disparate network protocols and internet architectures, to determine which is more resilient to various attack and under what circumstances. This may prove useful in motivating transitions to new technologies, and in optimizing the implementations of old ones.

We first present the core structure of our experiments in Section §5.1, including the minimal set of devices and their roles. We then discuss specifics of the network topologies on which our experiments are conducted and the DeterLab testbed environment in which those topologies are realized in Section §5.2. Section §5.3 discusses the selection of underlying performance metrics for evaluating the impact of DoS attacks. We then extend those metrics in Section §5.4 to define metrics for evaluating a DoS mitigation within a single specific context, and extend those further in Section §5.5 to analyze changes in results across multiple contextual variables. These metrics have been carefully crafted to address common shortcomings in the literature on DoS attack and mitigation measurement, as discussed in Section §3.3. They are agnostic to the selection of an underlying performance metric, control for both the separate and combined effects of an attack and a mitigation, and facilitate analysis across multiple other variables simultaneously.

5.1 Experiment Model

The minimal version of our experiment model involves three devices: a **server** provides some service or resource, a **client** attempts to access that service/resource, and an **attacker** attempts to disrupt that access. Our goal is to measure a mitigation's **efficacy** at reducing the attack's disruption, as well as the **overhead** it incurs at the client, server, or other entities outside periods of attack.

For certain experiments we may want to include multiple instances of each of these device types, perhaps adding additional attackers to increase the attack volume, or including multiple clients to assess fairness among them. We may also include additional device types providing other functionality – switches, routers, firewalls, etc.. While certain mitigations may be implemented on clients, servers, and/or attackers directly, others may be deployed on a separate device in the intervening network. More complex topologies may allow us to model real-world networks more closely, but they can also make it difficult to disentangle complex interactions between devices.

Figure 5.1 shows a simple example of the sort of network topologies on which we conduct our experiments. In addition to the server, clients, and attackers, it also includes a device we refer to as the **sink**. This represents the outside world – any packets in our network with destination IP addresses in the public Internet will be routed to the sink, where they are dropped immediately upon arrival. This allows us to safely observe the impact of the backscatter traffic that is sent in response to the spoofed-source attacks. Our attackers generate traffic with randomly selected source IP addresses (excluding private and reserved ranges), so any responses will be addressed to the rightful owners of those addresses. We cannot risk flooding any external devices with our attack backscatter, but that traffic may have a significant impact on devices inside our experiment as it shares network links and other resources with legitimate client traffic in just the same way the original attack packets do. In short, routing outbound packets to the sink before dropping them enables us to maintain realism without compromising safety.

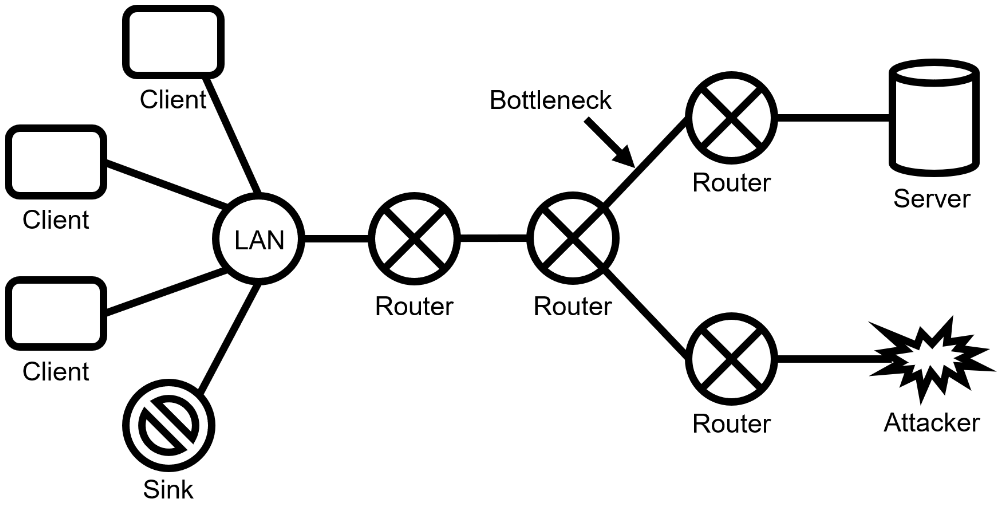

Figure 5.1: A simplified example of the sort of network topologies used in our experiments. Actual experiments use larger networks to improve realism, but this represents a minimal set of components.

Our essential topology also includes some set of routers and links to connect devices. In simple experiments these simply perform standard packet forward services one would expect, but they have additional potential for mitigation experiments. Some mitigations may be deployed on routers in the network rather than at the sender or receiver. As discussed further in Chapter §4, performing mitigation throughout the network (and particularly at the network edge, close to attackers) can allow us to drop malicious traffic earlier, minimizing the resources it consumes. Certain links in our topology are emulated, allowing us to easily adjust their bandwidth, latency, and loss between experiments. As shown in Figure 5.1 we use these emulated links to create artificial bottlenecks, which allow us to measure attacks and mitigations in diverse contexts. For instance, we may find that some mitigations are only effective when the bottleneck bandwidth exceeds the attack volume, or that their overhead scales with the bottleneck's latency in a way that makes them impractical for distant clients.

5.2 Testbed Environment

We now provide further detail on the selection of a testbed environment in which to construct our experiment topologies. The only accurate way to measure a volumetric DoS mitigation is by first generating a volumetric DoS attack, which presents obvious ethical concerns. We need an isolated environment in which the attacks we generate can only harm devices we control, without risk of leaking onto the public Internet. There are three main categories to choose from: simulation, emulation, or a physical testbed. Ideally we would like to experiment on a topology that is as similar as possible to some real-world system, but deciding between these options requires us to trade off realistic scale for realistic resource bottlenecks.

Testing in simulation with a program like ns-3 [80] would allow us to scale to thousands of nodes, but with each being a simplistic approximation of real hardware. Emulation with programs like Mininet [81] sacrifices some of that scale for added realism, but with virtual devices still sharing key system resources that could manifest as critical bottlenecks. Testing in an isolated physical testbed provides unparalleled realism in resources, but growing such a testbed to anywhere near Internet scale would be prohibitively expensive.

Resource bottlenecks are a key component of the approach to mitigation design we present in Chapter §4, and of our experimental methodology. As our results in Chapter 6 will demonstrate, subtle changes in resource availability can have a tremendous impact on the apparent impact of DoS attacks and the efficacy of mitigations. We therefore opt to conduct our experiments in a physical testbed, in order to maximize realism. Specifically, we have generously been given access to the recently updated DeterLab testbed, operated by the Information Sciences Institute (ISI) at the University of Southern California (USC) [82].

DeterLab was explicitly designed for this sort of network security research — in addition to isolating nodes from the public Internet, it also provides exceptional realism, flexibility, and reproducibility. We have effectively bare-metal access to testbed devices, which allows us to make arbitrary OS and kernel modifications that may be needed to implement mitigations, and also removes the risk of resource sharing with other users jeopardizing our experiments. However, we also have the option to create virtual machines and even combine the two in hybrid topologies. This lets us maximize realism where it matters most by using physical devices for the server and routers (where most bottlenecks occur), but leverage virtual machines to create larger numbers of clients and attackers (where bottlenecks are less significant). Devices are fully under our control while we have them reserved, but the DeterLab testbed as a whole is shared among many students and researchers – we cannot monopolize those resources, so we try to make the most of a small set of devices. For reference, the main topology used for our experiments in Chapter 6 consists of 7 bare-metal nodes and 20 virtual ones, though again the topology itself is a variable that must be considered in our methodology. Recent updates to DeterLab also provide support for Ansible [83], which allows us to fully automate the processes of configuring devices, generating traffic, and collecting data. In addition to simplifying the task of running experiments, this also helps improve the consistency of our results by ensuring we follow precisely the same steps each time. We can also easily share experiment "playbooks" with other researchers who may want to reproduce or expand on our work.

5.3 Performance Metrics

The starting point for all of our metrics is the selection of some underlying indicator of system performance. This could be a "legacy" metric like round-trip-time (RTT), throughput, gooodput, loss, or jitter; a resource metric like CPU utilization or network queue lengths; or something more complex.

Following recommendations from prior work by Mirkovic et al. [49]–[51] we rely primarily on transaction-oriented metrics derived from client-side application-layer data. These reflect the QoS experienced by end users more directly than so-called "legacy" metrics like RTT and loss can, since different applications may have drastically different resource requirements. For example, real-time audio applications like VoIP require low latency to prevent users from talking over one another, but need very little bandwidth and can tolerate a large amount of loss (at the human-layer if necessary, by simply asking "Could you please repeat that?"). High definition video streaming on the other hand requires high throughput to deliver its large amounts of data, but can tolerate significant latency by buffering before playback. If we want to know how a DoS mitigation will impact users we need to measure the applications they actually use.

Additionally, these client-side metrics tend to be much more lightweight than others. For a server to measure its CPU utilization it must expend some CPU cycles, and for a router to measure its bandwidth saturation it must interrupt packet processing. The act of measuring resource utilization increases resource utilization, in a way that cannot always be cleanly separated from the processes we actually want to measure. With client-side metrics we simply record a timestamp, attempt a transaction, then record the exit code and another timestamp. The overhead incurred from recording timestamps and status codes falls fully outside the time window of the transaction which prevents interference, and the resulting data requires clients to store only a handful of bytes per transaction which enables long-running experiments.

For these sort of transaction-oriented performance metrics, QoS can be defined as either the number of transactions a client can complete in a given period of time, or inversely as the average time it takes to complete a single transaction. Both may be useful to consider, and the way we capture data enables both to be extracted during analysis. Since our aim is to *maximize* QoS, we typically use the more intuitive framing of transactions-per-second (TPS).

It may be useful to track multiple underlying metrics and repeat our analysis process for each one. Transaction-oriented metrics are exceptional for determining *whether* a mitigation is effective, but resource metrics can provide more insight into *why* a mitigation behaves the way it does. Therefore our typical process is to start by looking at transaction-oriented metrics, and then examine resource metrics to help explain any unexpected results. Generally, one should select a set of metrics that is most relevant to the services being defended.

For example, to evaluate SYN Cookies which are intended to preserve a server's ability to accept new TCP connections, we can have clients repeatedly attempt to set up and tear down TCP connections, recording the start time, end time, and status for each attempt. Throughout this process we may also want to periodically record the number of open and half-open connections at the server, or the queue length and drop rate on some bottleneck link.

Once some (set of) performance metric(s) is selected, we can begin building more complex metrics from it. Again, the metrics defined in the following sections are agnostic to the selection of performance metric – they simply describe changes in *some* value.

5.4 Context-Specific Metrics

The core of our methodology is to measure each selected performance metric during four discrete experiments, controlling for both separate and combined effects of an attack and a mitigation. We refer to these experiments as follows:

- **UB** (Unmitigated Baseline): No attack, No mitigation
- **MB** (Mitigated Baseline): No attack, Mitigation
- **UA** (Unmitigated Attack): Attack, No mitigation
- **MA** (Mitigated Attack): Attack, Mitigation

We use these same acronyms to represent the average value observed for our chosen performance metric during the corresponding experiment. From these four values we can derive several further metrics for describing attacks and mitigations, as illustrated in Figure 5.2. First, we refer to the result of the UB experiment as the **baseline** QoS – this represents the ideal scenario. Deploying a mitigation incurs some (hopefully negligible) amount of overhead, defined as $UB - MB$. We can contextualize this value by normalizing it against the baseline performance, defining the mitigation's **percent overhead** as $\frac{UB-MB}{UB} * 100$

Next we define an attack's **threat** as $UB-UA$: the amount by which it reduces QoS when no mitigation has been deployed. Here too, it is helpful to consider the threat relative to the baseline. We define an attack's **percent threat** as $\frac{UB-UA}{UB} * 100$. An attack's **damage** is then defined as $UB - MA$: the amount by which it reduces QoS *with* the mitigation deployed. Again, the **percent damage** relative to the baseline is defined as $\frac{UB-MA}{UB} * 100$.

Lastly, we can define a mitigation's **efficacy** as the difference between the threat an attack poses and the damage it is able to realize: $(UB - UA) - (UB - MA) = MA - UA$. Normalizing this value is slightly more complex – in most cases we will want to measure efficacy as a percentage of baseline performance as above, but there is also value in viewing it as a percentage of the threat. As the threat approaches 100% these two interpretations will converge to the same value. When the threat is a small fraction of the baseline however, a completely successful mitigation will restore the same small fraction of performance relative to the baseline. In that case measuring efficacy as a percentage of the threat instead reveals that the mitigation is working as intended. The more standard interpretation of efficacy as a percentage of the baseline (and specific to some context C) is formalized in Equation 5.1:

$$\begin{aligned} E_C &= \frac{threat - damage}{baseline} * 100 \\ &= \frac{(UB - UA) - (UB - MA)}{UB} * 100 \end{aligned} \tag{5.1}$$

This single value of efficacy can represent data from arbitrarily many clients performing arbitrarily many transactions over arbitrarily long periods of time. Running experiments for longer durations provides greater statistical significance. There is still value in observing how underlying metrics change over time as is common in prior work, to measure the ramp-down and ramp-up in performance as an attack starts and ends, but we are interested in comparing across large sets of variables which requires abstracting away this detail. Our tests have shown that performance typically drops to a new steady state almost immediately upon the start of an attack, though the transition back to baseline performance is often longer. In the case of SYN floods, retransmitted SYN-ACKs may continue to flood the network for upwards of 30 seconds after the attack ends. It may be of interest to record the duration of these transition periods, but generally we exclude them to more cleanly compare steady-state performance across the four experiments.

Note that our efficacy metric alone is not sufficient for evaluation, it should always be presented in tandem with a mitigation's overhead. Only when combined can they provide an accurate indication of how practical a mitigation is for some specific context. In many contexts there is an inherent trade-off between them that must be evaluated to determine whether a mitigation is worth deploying: how much overhead can be justified outside periods of attack in order to protect against a potentially catastrophic threat? Yet the two may

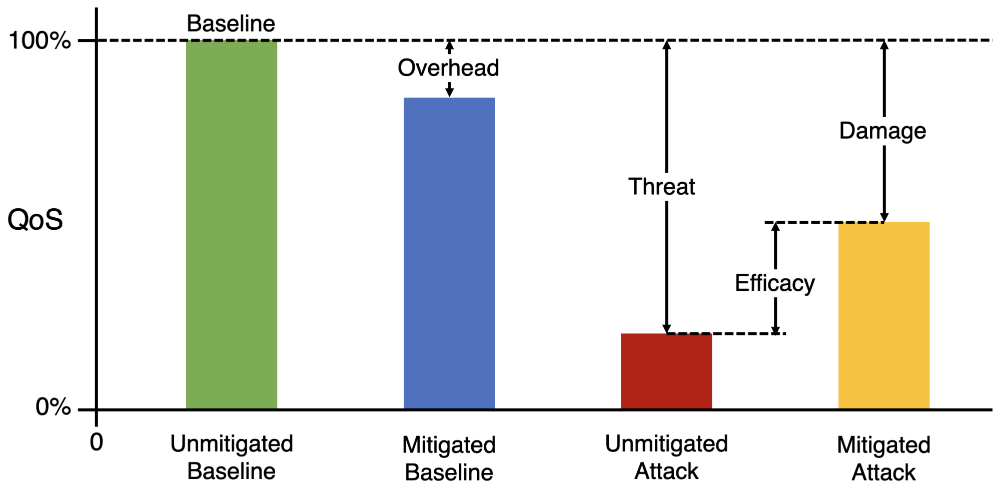

Figure 5.2: Our four experiments and key context-specific metrics.

depend on entirely different context variables, or depend on the same variables in different ways. We even need to evaluate trade-offs between different low-level performance metrics; for example, how should we value a mitigation that offers high efficacy in terms of client QoS but also imposes high monetary overhead, or one that improves fairness during an attack at the cost of increased energy consumption?

5.5 Cross-Context Metrics

There are innumerable factors that may constitute a meaningful change in context, including variations in attack type and rate, client application, network topology, device hardware, etc.. We divide these factors into two categories which lend themselves to different forms of analysis: **categorical variables** and **numerical variables**.

Categorical variables include client protocols, device hardware and operating systems, as well as certain aspects of the network topology. The simplest way to evaluate such factors is through a side-by-side comparison of results from separate sets of the four experiments described above. For example, say we want to compare SYN Cookie implementations in the Linux and BSD kernels. We run a full suite of experiments (with and without SYN floods being launched, and with and without SYN Cookies deployed) using a Linux-based server, then re-run the same experiments with a BSD server. It may be helpful to see raw data from these two scenarios side-by-side, or we can compute their difference to make statements like "the Linux implementation has $X\%$ higher overhead than the BSD implementation" or "the Linux implementation is Y times more effective than BSD's." We could compute a weighted average across multiple categories to make more general statements about a mitigation's utility, but assigning appropriate weights is challenging in many contexts, and the observation of an extreme outlier or a drastic difference between two categories is likely to be of more interest than a global average.

Numerical variables are those with a range of values (such as attack rate, client request frequency, and bottleneck link capacity), such that we can reason about increases and decreases in value. A general approach for any numerical variable is to plot it on the x-axis of a line graph with the mitigation's efficacy/overhead on the y-axis. Simultaneous analysis across two variables can be illustrated with a heatmap or by simply adding additional lines.

The rate/volume of attack is a particularly noteworthy numerical variable which warrants more complex analysis. By definition we expect the threat posed by a volumetric attack to depend heavily on this variable, which gives it universal importance within our scope. Additionally, since UA and MA experiments with an attack rate of zero would be identical to our baseline UB and MB experiments, we can simply compare the results of experiments as a function of attack rate, without needing to separately account for the mitigation's overhead.

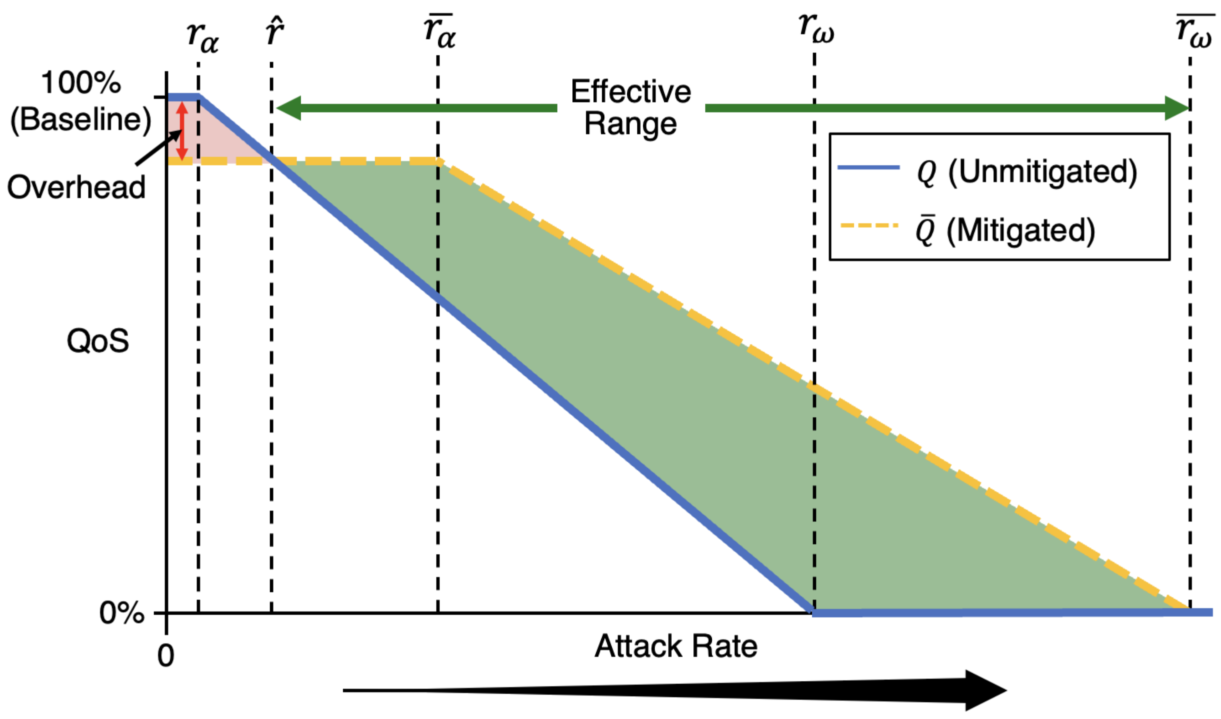

Figure 5.3: An example of how we might expect a DoS mitigation to change the way QoS depends on attack rate. This dependency can be defined by some function Q parameterized by the attack rate r, which morphs to function $\bar{Q}$ when a mitigation is deployed. Deploying a mitigation imposes some amount of *overhead* on the system, but hopefully improves QoS under some *effective range* of attack rates. Under extremely high rates of attack client QoS will approach zero regardless of the mitigation(s) deployed. We define the mitigation's *efficacy with respect to attack rate* as the area of the dark green shaded region to the bottom right of $\hat{r}$, minus the area of the light red shaded region to the upper left.

Consider Figure 5.3, which shows a simplified example of how client QoS (depicted as a percentage of the baseline QoS) typically depends on attack rate (Q), and how a mitigation may alter the function describing that dependence ($\bar{Q}$). There will always be some rate of attack r_α below which no measurable degradation of QoS can be observed – here a mitigation's efficacy is undefined since the threat is zero. There is also some rate r_ω above which the threat is so severe that client QoS effectively drops to zero, and a (hopefully higher) rate $\bar{r}_\omega$ at which QoS is zero even with the mitigation in place (because the system is so congested with attack traffic that client requests never reach the server). Between the extremes of r_α

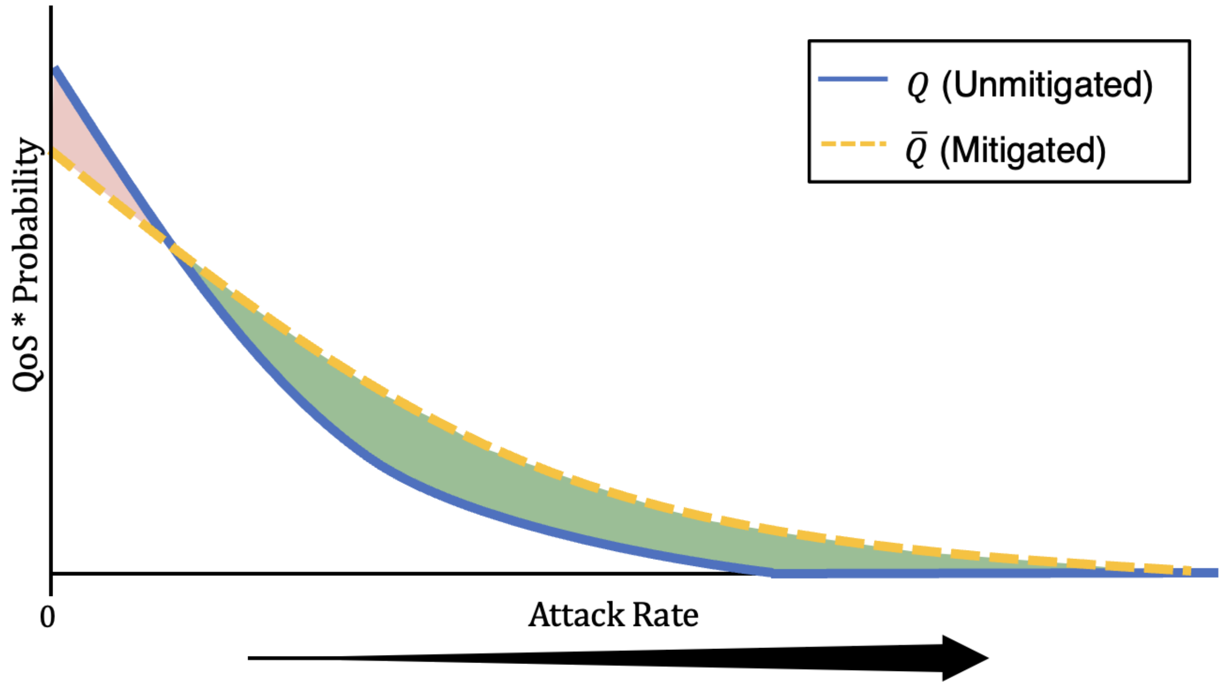

Figure 5.4: An example of how our perspective of Figure 5.3 might be skewed by a long-tail probability distribution of attack rates. Higher rates of attack are significantly less likely to occurr, meaning we should place less weight on their effects. By multiplying Quality of Service by probability of attack on the y-axis, we see that this distribution augments overhead from no/low-rate attacks and diminishes efficacy from high-rate attacks.

and $\bar{r}_\omega$ exists some rate $\hat{r}$, the lowest attack rate for which a mitigation's benefits begin to outweigh its overhead.

The mitigation's **efficacy with respect to attack rate** is visualized as the area of the dark green shaded region to the bottom right of $\hat{r}$, minus the area of the light red shaded region to the upper left. More formally, we can define this as:

$$E(r) = \int_0^{\bar{r}_\omega} \bar{Q}(r)\,dr - \int_0^{\bar{r}_\omega} Q(r)\,dr \tag{5.2}$$

We refer to the interval $(\hat{r}, \bar{r}_\omega)$ as the mitigation's **effective range**. Ideally a mitigation should *only* be deployed when attack rates are in this range – for lower rates the mitigation's overhead will make it counter-productive, and for higher rates it will have no effect. Yet in practice it may be infeasible to toggle a mitigation on and off precisely when attack rates cross these thresholds. Some attack traffic is indistinguishable from legitimate traffic which makes it impossible to determine the attack rate, and toggling some mitigations on and off may require delays or even device resets.

In these cases, we need some way to determine whether a mitigation is worth enabling permanently (or at least for some extended period of time that sufficiently amortizes the cost of toggling). The first step is to identify the distribution of attack rates that the server *expects* to receive. This is an extremely challenging problem requiring longitudinal data about past traffic spikes for some context of interest. In most cases we expect to see a long-tail distribution, with no attack the vast majority of the time, somewhat frequent small-scale attacks, and very rare instances of extremely high-volume attacks.[1]

After defining an expected distribution, we multiply the two functions in Figure 5.3 by it. We then integrate the resulting functions and take their difference to compute the mitigation's **adjusted efficacy with respect to the expected attack rate**. More precisely: let $Q(r)$ be the function describing QoS with respect to attack rate (r) without the mitigation deployed; let $\bar{Q}(r)$ be the function describing QoS with respect to attack rate (r) with the mitigation deployed; and let $P(r)$ be the function describing the probability distribution of attack rates. Then the adjusted efficacy with respect to expected attack rate distribution is defined as:

[1]Note that this is from the perspective of a single device, making local decisions about its own mitigation strategies. Looking Internet-wide there are almost always multiple ongoing DoS attacks at any given time.

$$\widetilde{E}(r) = \int_0^{\bar{r}_\omega} P(r)\bar{Q}(r)\, dr - \int_0^{\bar{r}_\omega} P(r)Q(r)\, dr \tag{5.3}$$

Assuming $P(r)$ is a long-tail distribution as described above, this formulation will appropriately emphasize the overhead imposed by the mitigation in the common case where no attack is present, and de-emphasize benefits the mitigation provides in rare instances of high-volume floods.

This adjusted efficacy is perhaps the best single-value indicator of whether a mitigation is worth deploying across the contexts of differing attack rates, but still only applies to a single broader context (a single network topology, client application, etc.). After computing $\widetilde{E}(r)$ and/or other complex metrics describing the mitigation's relation to attack rate, we can return to our simpler approaches to cross-context analysis. For instance, how does the mitigation's effective range depend on bottleneck link capacity? How do its $\hat{r}$ and $\widetilde{E}(r)$ values depend on the choice of client application? Answering such questions is vital to understanding when and where a mitigation should be deployed, though a full understanding will require a tremendous amount of experimentation.

5.6 Summary

In summary, this chapter has presented an experimental methodology and set of metrics for more accurately evaluating DoS attacks and mitigations. Our experimental topologies are microcosms representing small portions of the Internet. A small set of clients attempt to communicate with a single server, while a small set of attackers attempt to disrupt them. Crucially, we route attack backscatter (generated in response to spoofed-source attack packets) through key bottlenecks before dropping it for safety. We use a physical testbed to maximize realism of critical resource bottlenecks in the system, but also leverage emulation for select devices to increase the size of our network.

Our metrics are designed to work with arbitrary performance metrics, describing how they change in response to the effects of attacks, mitigations, and various context variables. We discussed how to accurately measure a mitigation's efficacy and overhead, why the two should be considered as a pair, and how to interpret the relation between them based on the likelihood of attack. In the following chapter (§6) we apply this measurement framework to evaluate the novel mitigations designs we presented in Chapter §4.

Chapter 6

Empirical Analysis

Here we present our experimental analysis of the mitigations described in Chapter §4, following the measurement framework defined in Chapter §5. Section §6.1 discusses the topology on which our experiments are conducted.

Before evaluating our own mitigations, we first provide an examination of SYN Cookies in Section §6.2. As discussed in §3.2.3, they are broadly deployed in the wild despite a lack of standardization and a dearth of empirical analysis. While not explicitly designed to combat high-volume attacks, it is important to understand exactly where and how SYN Cookies fail under such conditions. They also inform what we might hope to see in a more effective mitigation, providing a useful reference point against which to compare our other results.

Next we present our evaluation of SYN Padding in Section §6.3. This extremely simple mitigation provides similar results as SYN Cookies when attackers fail to participate, with the notable advantage that verification can be dislocated from the server to enable earlier packet drops and distributed verification. This is a step in the right direction, but SYN Padding also has two key shortcomings: if attackers do participate and pad their packets it has low efficacy (unless they are bandwidth constrained which we expect is rare); and it has poor scalability since the amount of padding is limited by the maximum packet length.

Both of these issues are overcome by our second, more robust mitigation SYN PoW, evaluation of which is presented in Section §6.4. It protects against attackers whose traffic volume is limited by CPU resources, which we believe to be the common case, and offers a high degree of freedom in tuning proof difficulty to increase efficacy at the cost of higher overhead. It also provides the same portability of verification and stateless operation as SYN Padding, giving it a clear advantage over SYN Cookies.

6.1 Experiment Topology

Figure 6.1 illustrates the topology on which our experiments are conducted, following the general experiment model presented in §5.1 and realized in the DeterLab network testbed as discussed in §5.2. Recall that our overall approach is to measure the ability of clients to complete transactions with the server, controlling for the separate and combined effects of an attack and a mitigation. We divide clients into four subnets, labeled A-D in order of their proximity to the server. We refer to subnets A and B on the near side of the bottleneck link as "**local**", and to subnets C and D on the far side as "**remote**". As our results will illustrate, the relative location of devices can have a tremendous influence over baseline performance, attack impact, and mitigation efficacy. Local clients tend to fare significantly better than remote ones, maintaining service under higher-volume floods. We divide attackers across subnets B and D, but primarily rely on the subnet D attackers since only they are constrained by our bottleneck link. The Sink node (to which attack backscatter traffic with public destination

IP addresses is routed before being dropped) is located in subnet C, and the server is in subnet A.

While we have a high degree of freedom in adjusting the bottleneck link's capacity, its latency is held constant at 1ms across all experiments. We acknowledge that this is significantly lower than that of most real-world networks, but this is an unfortunate limitation of our testbed environment. The physical devices of our testbed are all in very close proximity to one another (within the same data center), and adding artificial latency requires some intermediary device to buffer packets for the duration of that latency. Under high-volume floods this buffering quickly becomes untenable. As a concrete example, adding 100 ms of latency during a 50 Mbps flood would require at least a 5Mb packet buffer. Since SYN-ACKs sent by the server in response to the SYN flood also transit the bottleneck link, the actual requirement is double that buffer capacity. In our attempts at conducting higher-latency experiments this buffering caused inconsistent traffic rates at best, and often resulted in full crashes of the network emulation tool which makes any data collected unusable and requires an experiment to be restarted from the beginning. To prevent the emulation tool from becoming the primary bottleneck, we limit ourselves to low-latency experiments for the time being. As discussed in §7, future experiments in simulated topologies could overcome this hurdle, at the cost of a reduction in realism.

Even though remote clients do not experience what would typically be considered high latency, their RTT to the server may still be up to ten times that of local clients. We measure this baseline latency between the server and one client from each subnet[1], sending 100 sequential pings from each at a 0.1 second interval (well above the maximum latency). Table

[1]All clients within the same subnet exhibit nearly identical behavior.

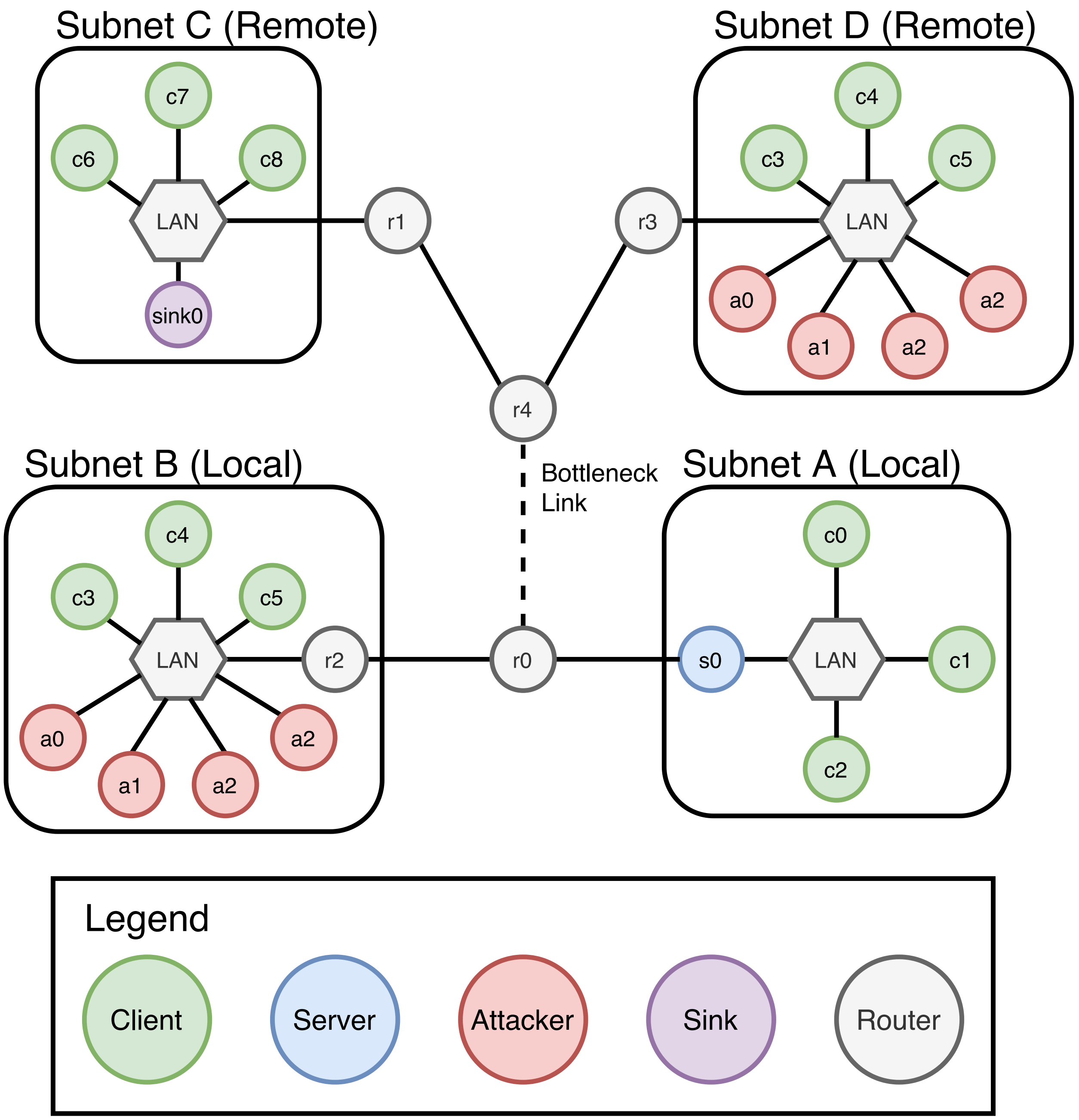

Figure 6.1: The network topology used in our experiments. Nodes are named with an abbreviation of their role: **s** for server, **c** for client, **a** for attacker, and **r** for router. The labeled bottleneck link between routers r0 and r4 can be constrained using network emulation. By default it is set to 1Gbps capacity, 1ms latency, and 0% loss. We label the four subnets A through D to help distinguish results from clients in different locations. We often observe drastically different behavior between these subnets, particularly when comparing the two local subnets (A and B) against the two remote ones (C and D).

6.1 illustrates the mean, minimum, maximum, and standard deviations we observe. Remote clients are below 3ms on average which is objectively quite fast, but still slow compared with 0.237 ms for the client in Subnet A and less than double that for the client one hop away in subnet B. No packet loss was experienced during these baseline latency measurements.

As our results in the remainder of this chapter illustrate, these difference in latency are sufficient to observe significant differences in client performance, attack impact, and mitigation efficacy. If all latencies were multiplied by a constant factor to reach more realistic values (ie. a minimum RTT of 2.28 ms for subnet A, 4.0 ms for subnet B and 26.8 ms for subnets C and D, if multiplying all by a factor of 10), we would expect to see the same general patterns distinguishing local and remote clients. The locations of devices relative to the attack and the ratio of their latencies have a greater impact than the absolute value of their latencies.[2]

Subnet	Client	Min RTT (ms)	Mean RTT (ms)	Max RTT (ms)	σ (ms)
A	c0	0.228	0.273	0.329	0.013
B	c3	0.400	0.434	0.613	0.021
C	c6	2.682	2.796	3.194	0.114
D	c9	2.682	2.859	3.282	0.143

Table 6.1: Baseline latencies between each client subnet and the server. For one client in each subnet we perform 100 pings with a 0.1 second interval and report the minimum, mean, maximum, and standard deviation. Local clients in subnets A and B observe average RTTs of <0.25ms and <0.5ms respectively. Remote clients in subnets B and C also have very fast connections with <3ms average RTT, yet this is an order of magnitude greater than for clients in subnet A. No packet loss is observed.

[2]Within reason. Under extremely high latency applications will begin to fail completely. Our clients fail and retry after a one second delay, so baseline performance would drop to zero with an RTT of $\leq$ 1000 ms.

6.2 SYN Cookies

We begin with an examination of SYN Cookies in isolation, in order to examine how they fail under high-volume SYN Floods, and to provide a point of comparison for our later analyses of SYN Padding and SYN PoW. We conduct experiments on a subset of the topology depicted in Figure 6.1. We measure attacks launched from 1, 2, 3, and 4 attackers, all in the remote set of nodes a4-a7, in order to observe the effects of increasing attack volume. Each attacker is limited to 10% of its available CPU resources using the `cpulimit` utility. This reduces the maximum threat, which allows us to observe the impact of increasing attack volume more clearly – with 100% CPU resources the threat is so severe that the mitigation fails completely on remote clients (although it is near 100% effective on local ones).

In all experiments, we have the full set of twelve clients attempt to communicate with the server, and we separately test the following three client/server applications. Typical packet exchanges for each application are depicted in Figure 6.2.

- **TCP Setup**: The client initializes a TCP connection with the server and then immediately, but cleanly tears it down. This ideally results in the following five-packet exchange, as depicted in the left side of Figure 6.2: SYN, SYN-ACK, ACK, FIN-ACK, ACK for each transaction. While we expect most real-world TCP transactions will be longer connections involving some exchange of data[3], this minimal transaction provides a good indicator of a server's availability in terms of its ability to open new connections. This is the primary resource SYN Cookies are designed to protect.

[3]It's worth noting that clients can piggyback an early data payload in their ACK packet. So while it may be uncommon, this simple five-packet exchange does have a practical use.

- **HTTP 1KB**: The client requests a 1KB file from the server over HTTP using the `curl` command. The file's contents are randomly generated data, and caching is disabled to ensure independence between transactions. Under normal conditions, a file of this size can be transferred using two data packets (PUSH-ACK), as depicted in the right side of Figure 6.2.

- **HTTP 1MB**: The client requests a 1MB file from the server over HTTP, as above. A file of this size requires many more packets to transfer, though the exact number is surprisingly variable. In the absence of any other traffic, we've observed the same client-server pair require anywhere from 31 to 46 data messages to complete a 1MB transfer. Each PUSH-ACK will prompt at least one ACK in response, unless lost in which case it will be retransmitted. Some ACKs may also be lost, especially those from remote clients suffering higher latency, which will also be retransmitted. The entire exchange may exceed 100 packets, of which notably only one is a SYN.

Figure 6.3 illustrates how baseline performance differs across these applications, and how it depends on client location. Each dot represents a single client device with a single set of parameters. As discussed in §6.1 above, we refer to the clients on the near side of the bottleneck link to the server (c0-c5) as "local" and those on the far side (c6-c11) as "remote". As expected we observe significantly higher baseline performance for local clients, and for simpler client applications. In several cases we see two clear bands emerge for the same application in the same location; this is explained by the difference between the two subnets on the same side of the bottleneck, with subnet A slightly outperforming B, and C slightly outperforming D. Local clients are able to complete over 400 of the simple TCP Setup

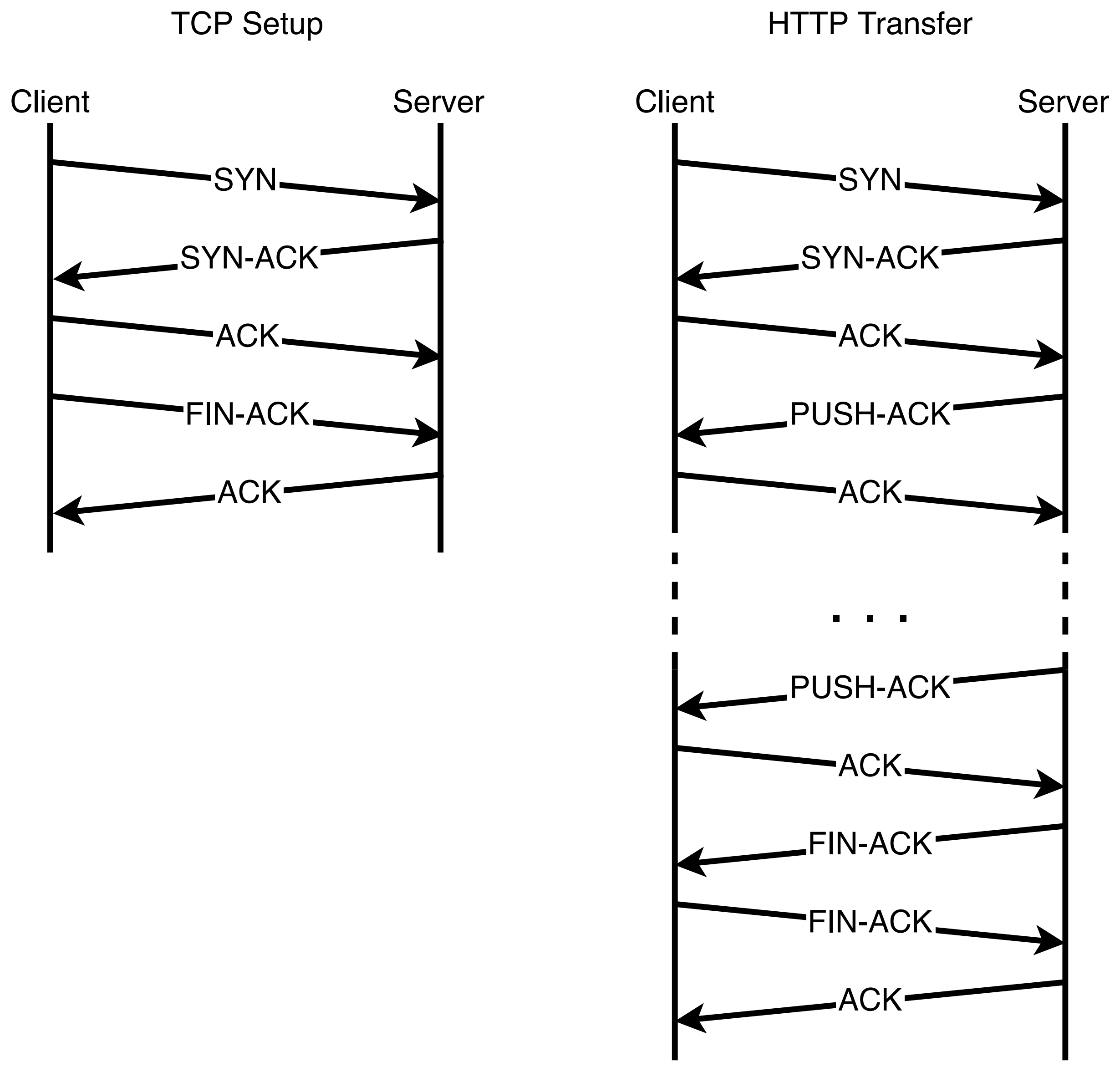

Figure 6.2: The packet exchanges that typify our client applications. The TCP Setup client initiates a connection and then immediately (but cleanly) tears it down in a 5-packet exchange. The HTTP clients retrieve files from the server using some number of PUSH-ACK packets. Transferring a 1KB file typically takes two PUSH-ACKs (and two corresponding ACKs) as depicted, while larger files will take more. A 1MB file may take $\sim 30 - 50$. In longer exchanges, the server will often send several PUSH-ACKs in a burst (as its window size permits) followed by an equal burst (assuming none are lost) of ACKs from the client, rather than the consistent alternation depicted here.

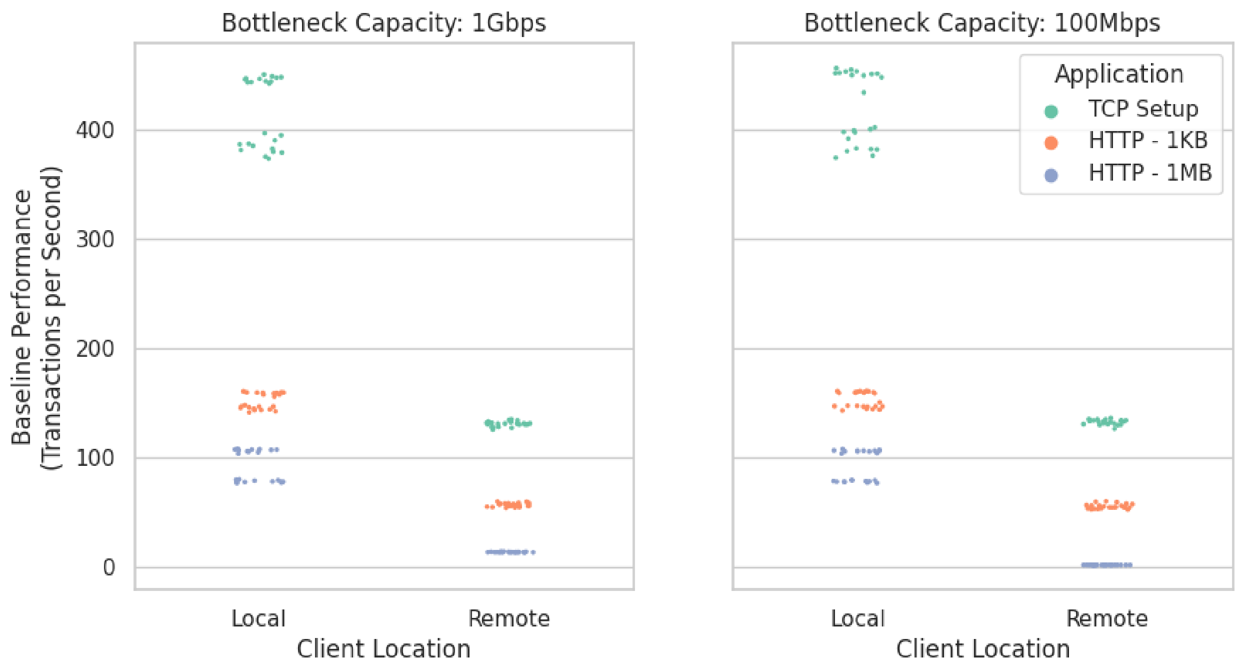

Figure 6.3: Baseline performance based on client application and location, for our analysis of SYN Cookies. As before we see better performance for clients that are local to the server, and for those running simpler applications requiring fewer round trips per transaction. Constraining the bottleneck link from 1Gbps to 100Mbps has little impact except for remote clients running the HTTP 1MB application, which has significant throughput requirements. Their already slow service drops to near zero.

transactions per second, while remote ones complete only ~ 150. The more complex HTTP applications fare significantly worse in both cases because they require more round trips to complete. Remote clients often reach the 1 second timeout before they are able to complete a 1MB transfer, resulting in a baseline near zero.

It is worth noting that in separating results based on bottleneck link capacity we see surprisingly little difference between the 1Gbps and 100Mbps contexts, except in the case of the 1MB HTTP transfer application. That application has significantly higher throughput requirements than the others, and so its performance unsurprisingly drops to near zero when the bottleneck is constrained. Local clients should not and do not exhibit any differences, since their traffic never transits the bottleneck.

Next we evaluate the overhead of SYN Cookies on client performance outside periods of attack. Figure 6.4 shows that the mean overhead is near zero across all contexts. The variance we observe can be accounted for by the variance in baseline performance. If performance happens to be slightly below average when the baseline is measured and slightly above average when overhead is measured then overhead will appear negative, and vice versa. Relative to these natural fluctuations clients experience in performance over time, overhead is negligible from the application-layer perspective. The one extreme outlier of 100% overhead for a local TCP Setup client in the 1 Gbps context is the result of a crash fault on a client device, not an effect of SYN Cookies.

Note that overhead is presented here as a percentage, in order to normalize it to the baseline performance. A measurement of 10% overhead could indicate a decrease of nearly 50 transactions per second (for local clients running the TCP Setup application), or as little as a fraction of a transaction per second (for remote clients running the HTTP 1MB application).

Next we measure the threat posed by the SYN flood attack, without the mitigation deployed (UA). Service is completely disrupted for both HTTP clients with a threat of 100%, while the simpler TCP client manages to complete a few occasional transactions. Even a single attacker node operating with just 10% of its CPU resources is able to degrade service to just $\sim$ 4% of baseline performance with an unconstrained (1 Gbps) bottleneck link.

Finally, we examine the efficacy of SYN Cookies at mitigating this devastating attack, with results depicted in Figure 6.6. We see that local clients consistently experience efficacy near 100%, with the more complex HTTP 1MB application lagging slightly behind the

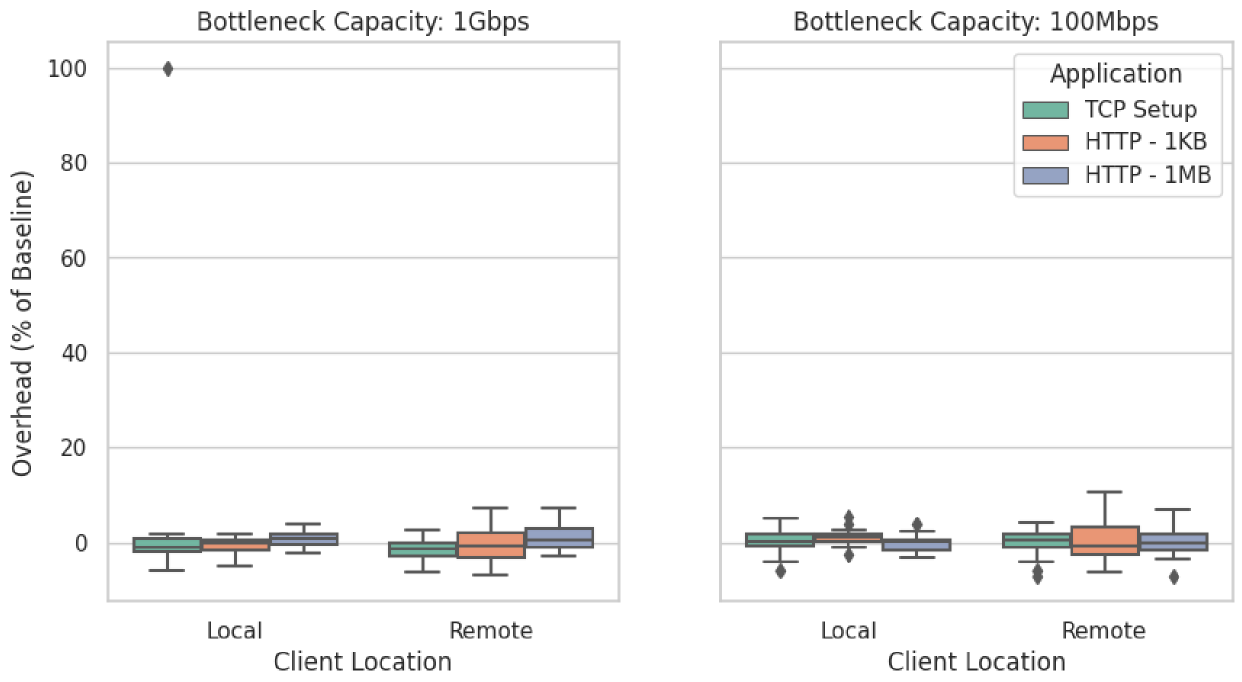

Figure 6.4: The overhead of deploying SYN Cookies at the server. It is negligible relative to variance in the baseline, with a mean near zero regardless of client location, client application, or bottleneck capacity. The one extreme outlier of 100% overhead represents an unrelated device failure, not an effect of SYN Cookies.

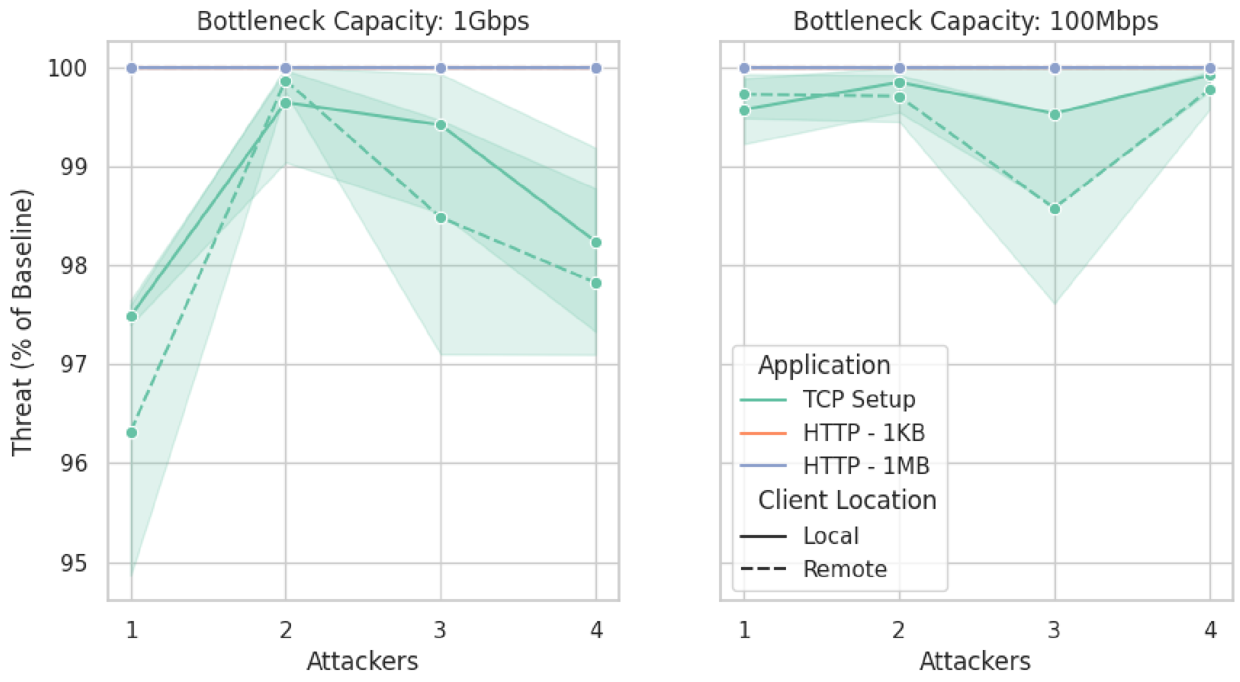

Figure 6.5: The threat posed by our SYN flood attack, for our analysis of SYN Cookies. HTTP client applications experience a complete disruption of service with 100% threat, while the simpler TCP Setup client manages to complete occasional transactions but at a greatly reduced rate. Bottleneck capacity, client location, and number of attackers all have little impact, as even the least potent attack is still devastating across all contexts.

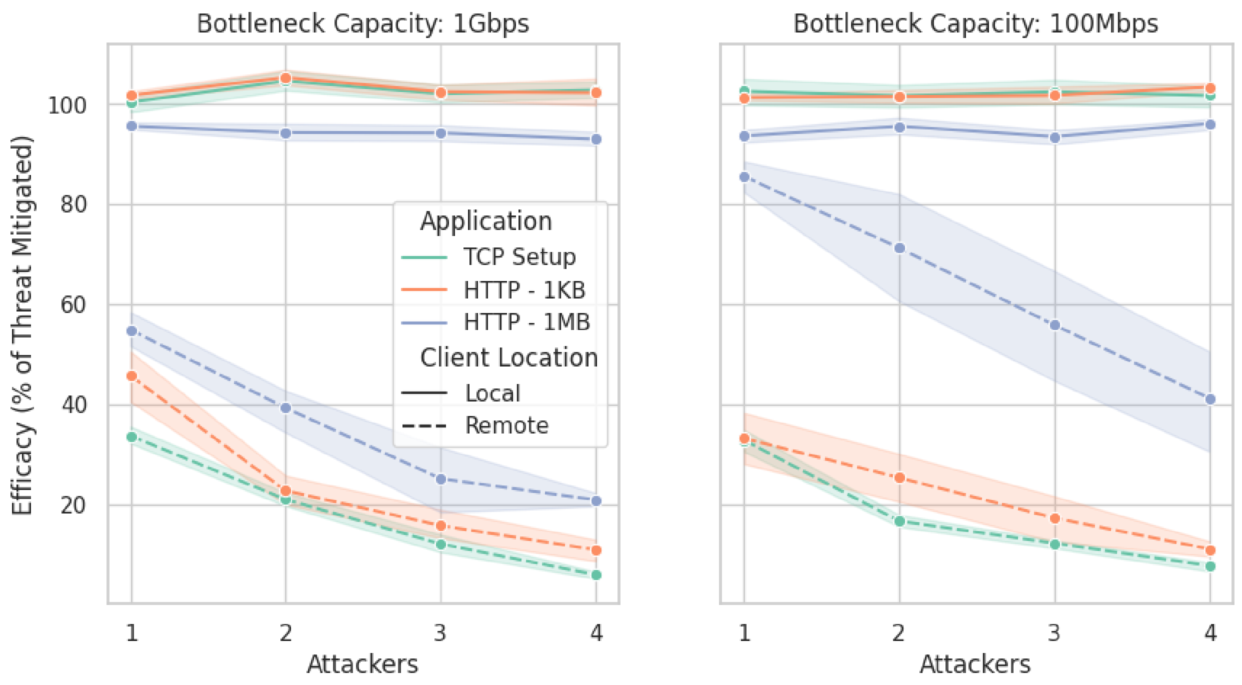

Figure 6.6: The efficacy of SYN Cookies at mitigating high-volume SYN Flood attacks. Local clients experience near 100% efficacy, while remote ones see a steady decline as the number of attackers increases.

other two. For remote clients this trend is inverted, with the more complex applications experiencing higher efficacy. We also see the clearest impact of attack volume in remote clients, with a steady decline in efficacy for all applications as the number of attackers increases. The most significant difference between the 1 Gbps and 100 Mbps bottleneck is the large increase in efficacy for remote HTTP 1MB clients when the bottleneck is constrained. This follows from the drop in baseline performance observed for those same clients in Figure 6.3. When the baseline is very low the efficacy tends to be higher – it is typically much easier for a mitigation to restore performance from 0 to 1 transactions per second than from 0 to 100.

These results clearly show that the efficacy of SYN Cookies scales poorly with attack volume, at least for clients whose traffic must share a bottleneck link attackers. Again, SYN

Cookies were not explicitly designed to protect against such high volume floods, but it is valuable to see exactly how their efficacy declines as the flood's volume increases. Since most real-world clients are likely positioned more similarly to our remote clients than our local ones, we expect SYN Cookies offer insufficient protection under volumetric attacks. Even our most powerful flood with four attackers is paltry compared to actual attacks on the modern Internet.

Fortunately, the overhead of SYN Cookies is low enough that leaving them deployed during such floods is unlikely to cause harm, though since each SYN packet requires a fixed amount of work to generate a cookie, there is a possibility that even larger attacks would result in more significant overhead, and adding a large number of additional clients could have the same effect. Resource limitations of the testbed we use preclude such large-scale experiments, but we encourage future work (either in a larger testbed or in simulation) to confirm how the overhead of SYN Cookies scales under extremely high-volumes of traffic, be it malicious or legitimate.

In summary, while leaving SYN Cookies deployed against high-volume SYN floods is unlikely to cause harm, it is also unlikely to provide adequate protection for most clients. This further strengthens the case for developing new mitigations that are tailor-made to combat volumetric attacks, such as our SYN Padding and SYN PoW proposals.

6.3 SYN Padding

Here we present the evaluation of our SYN Padding mitigation, defined in Section §4.2.2. We first explore its objective ability to reduce the volume of a SYN flood in Section §6.3.1, followed by its subjective overhead and efficacy from the client perspective in Section §6.3.2. We note that SYN Padding is intended primarily as a toy example to illustrate our TIPs approach, and as a thought experiment to explore the relation between packet length and DoS resilience. While we achieve promising results in certain contexts, we would not necessarily recommend deploying SYN Padding anywhere as presented – rather our hope is that these findings will inform the protocol design process going forward, and perhaps trigger a re-examination of *all* minimum packet lengths at the network layer (in IPv4 and IPv6).

6.3.1 Traffic Rate/Volume Reduction

We begin our analysis of SYN Padding by measuring the amount by which it is able to reduce the volume of traffic produced by a single attacker arriving at the server. These initial experiments are performed on the topology depicted in Figure 6.1, but using only a small subset of the devices. For each set of parameters tested, a 30 second SYN flood is launched from node a5 with a destination of node s0. The SYN Padding verifier is deployed on the ingress side of node r0's network interfaces, with varying proof thresholds. Node s0 records the average attack **rate** in bits per second and the average attack **volume** in packets per second, using the `tcpstat` utility. This tool outputs one data point for every 5 second window – we exclude the first and last data points for each trial as their windows include time before the attack starts and after it ends. We then average results from the middle 20 seconds which represent the real steady-state nature of the attack.

We begin with the bottleneck link set to 1Gbps, which is not actually expected to be restrictive. In this case the attacker's maximum possible bitrate (without SYN Padding) is

61.43 Mbps (166.94 Kpps). Deploying SYN Padding here actually proves counterproductive, keeping the flood's packet volume relatively constant at 168.22, but *increasing* its rate to 107.66 Mbps. This is not surprising, since our starting assumption is that most attackers are naturally CPU-bound, and we expect SYN Padding to be effective only when the primary bottleneck for the attacker is its outgoing bandwidth. That means for best results we need to constrain that bandwidth to below its maximum rate of 61.43 Mbps. To verify this we next repeat our measurements of the attack rate and volume, with and without SYN Padding, this time using DeterLab's network emulation utilities to artificially restrict the bandwidth of the bottleneck link between routers r4 and r0. Values tested are $\in \{500, 100, 75, 50, 25, 10\}$ (Mbps).

Our assumption is that per-packet overhead tends to dominate in volumetric floods, at least for in-network devices. Each packet forces forwarders to perform routing table lookups, update TTL fields, and compute checksums, all of which incur fixed costs regardless of the packet's contents.[4] Figure 6.7 shows the impact of SYN Padding on attack volume (in Kpps). We see that padded packet floods are *always* lower volume than unpadded ones, except when the bottleneck is completely open at 1Gbps, in which case the two are equivalent. As expected, the benefit is greatest when the bottleneck is reduced below the attacker's natural sending rate of 61.43 Mbps. We draw dotted black lines between padded and unpadded points at the same x-value to aid with visual comparison. In summary, if (as we assume) the costs of handling a flood are are primarily per-packet rather than per-bit, then packet

[4]Admittedly, the effort required to compute a checksum does depend on packet length, but to a lesser extent than application-layer operations that receivers typically perform. For example, a packet that contains a large file and asks the receiver to compute a diff between it and some other existing file would incur predominantly per-bit overhead. Recall that volumetric attacks typically comprise extremely simple packets which contain no such application-layer payloads.

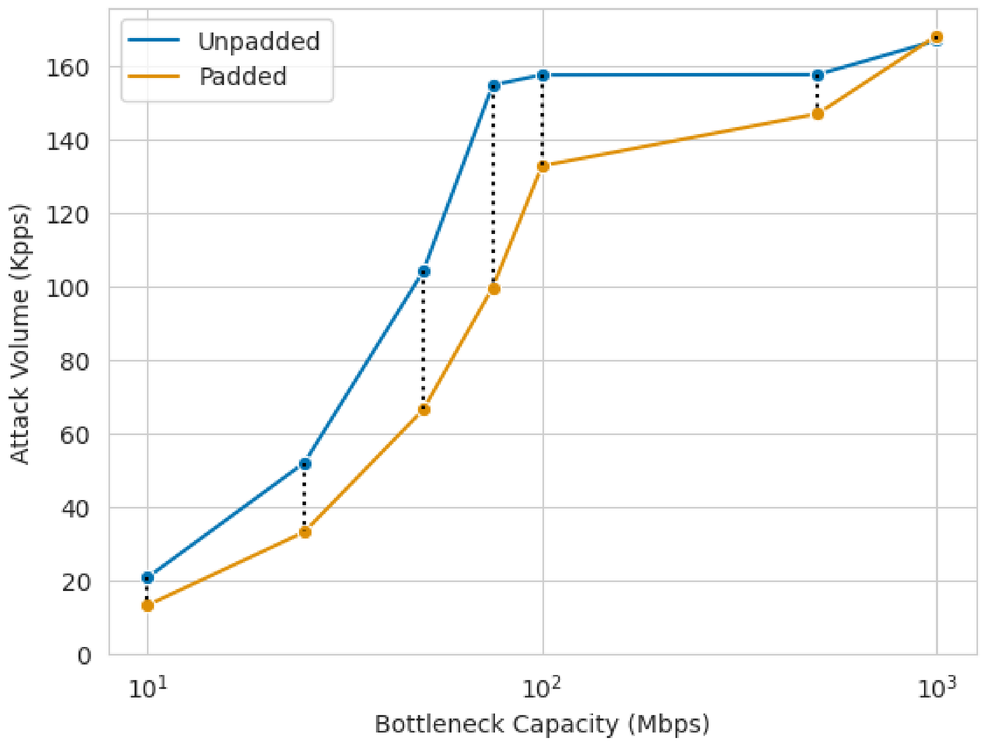

Figure 6.7: The change in attack volume (in terms of thousands of packets per second) caused by the addition of SYN Padding. Dotted black lines are drawn to connect points at the same x-value in order to help visualize the difference.

padding is a promising strategy for mitigation.

The trade-off for reducing attack volume in this way is an increase in attack rate. In a padded flood attackers send fewer, longer packets. Again, we assume the benefits of receiving *fewer* packets outweigh the detriments of receiving *longer* packets, but it is important to examine how SYN Padding impacts an attack's bitrate in addition to its volume. If attackers truly are bandwidth-constrained the attack rate should remain unchanged at their sending limit. If not, their floods will contain more bits with the same amount of packets, causing the mitigation to be somewhat counterproductive.

Figure 6.8 shows attack rate (in Mbps) as a function of bottleneck capacity, both with and without SYN Padding. The dashed black line depicts the equation $y = x$, representing the maximum possible rate that could be sent if the attacker were to fully saturate the bottleneck link. As expected, we see that both padded and unpadded attacks closely approximate this ideal rate as long as the bottleneck is constrained to below the attacker's natural (CPU-bound) flooding rate. Once the bottleneck link capacity is loosened above that threshold, we see the attack rate flatten as the primary bottleneck instead becomes the attacker's CPU. The unpadded attacker appears CPU-bound with a bottleneck capacity of 75 Mbps and above, indicating that its tipping point must be somewhere between that value and the next lowest of 50 Mbps, which matches expectation based on the 61.43 Mbps maximum rate. The padded attacker becomes CPU-bound with a bottleneck capacity of approximately 100 Mbps, again matching expectation based on its 107.66 Mbps maximum rate. We attribute the slight uptick in both rates for extremely high bottleneck capacities to imperfections in the network emulation tool. This also explains the slight gap observed between the real and ideal rates when attackers are bandwidth-constrained.

Figure 6.9 provides another perspective on this same data, combining information from Figures 6.7 and 6.8. Here the x-axis again indicates the bottleneck link capacity, while the y-axis now shows the *percent change* in attack rate/volume (with rate represented by the blue line and volume by the orange). The vertical dashed line again indicates the attacker's natural flooding rate of 61.43 Mbps. Here we see that the attack's rate is always increased by padding (positive percent change) while its volume is almost always decreased (negative percent change). As expected, we observe the highest volume reduction when the bottleneck capacity is at or below the attacker's maximum rate. For values of 75 Mbps and below SYN Padding provides a consistent 36% reduction in the number of packets the server receives (a -36% change in attack volume). Note that this threshold is slightly higher than the

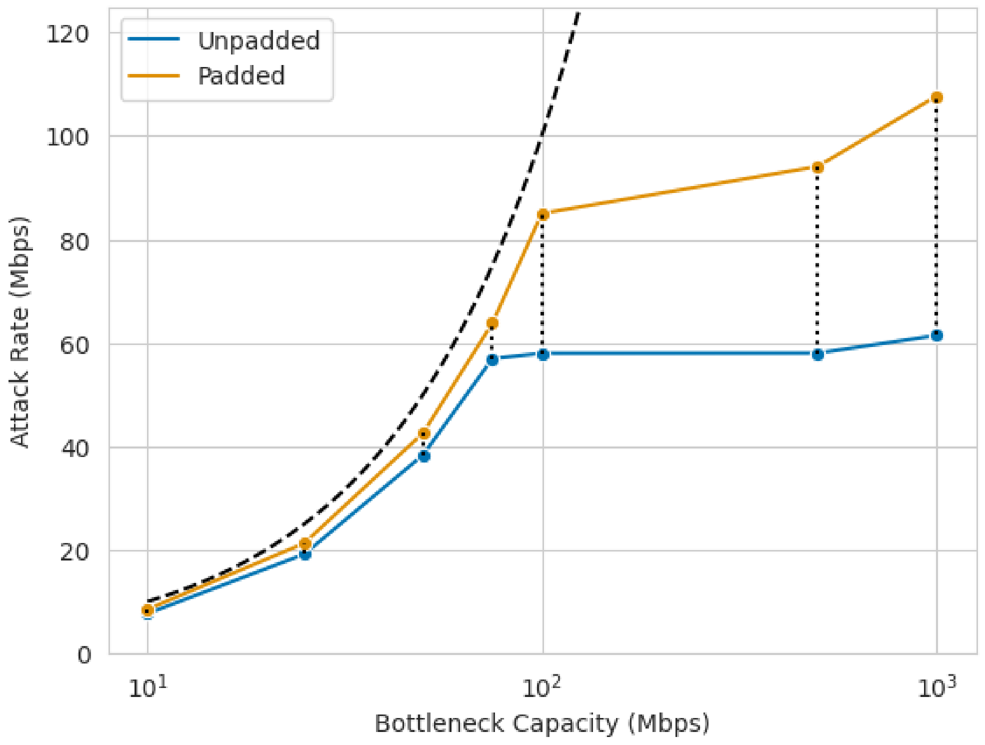

Figure 6.8: The change in attack rate in terms of Mbps caused by the addition of SYN Padding. Dotted black lines are drawn to connect points at the same x-value in order to help visualize the difference. The dashed black line shows the equation y=x, representing the maximum possible attack rate that could get through the bottleneck.

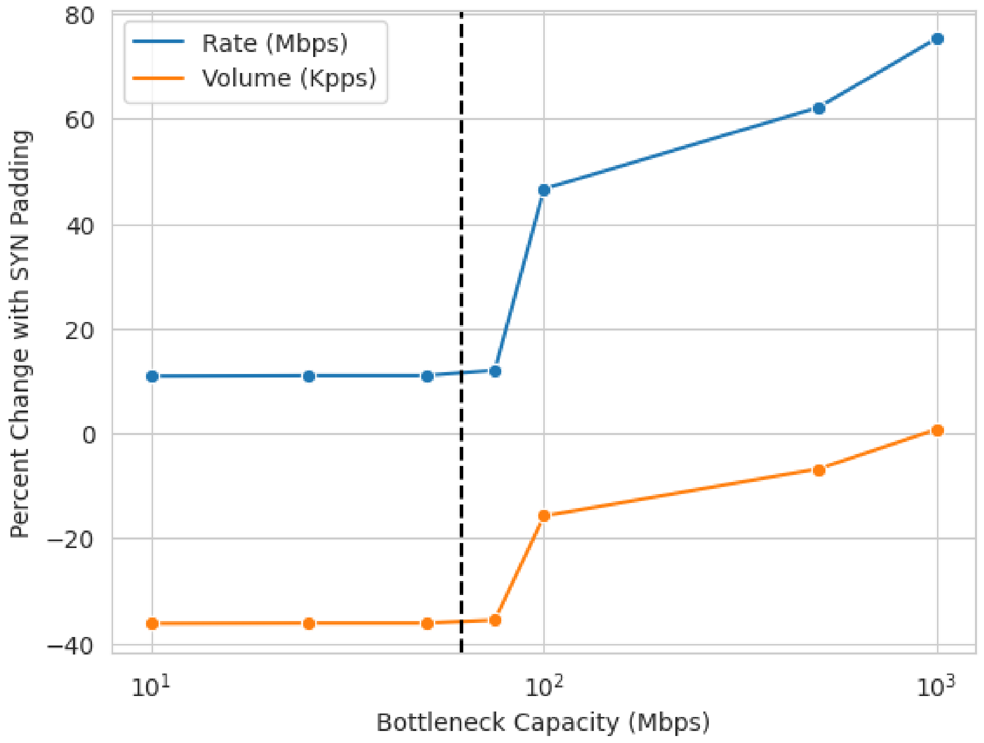

Figure 6.9: The percent change in attack rate and volume caused by the addition of SYN Padding. The vertical dashed line indicates the attacker's maximum attack rate without padding when the bottleneck is unconstrained. As long as the bottleneck capacity is at or below this value (or very slightly above it), SYN Padding succeeds in reducing the attack volume by approximately 36%, at the cost of a roughly 10% increase in the attack rate.

attacker's actual maximum rate – looking at Figure 6.8 this is not surprising, as we can see the maximum rate received is always at least slightly less than whatever bottleneck capacity is set – a bottleneck of 75 Mbps results in an actual unpadded attack rate of just 56.97 Mbps, which is indeed below the unconstrained maximum of 61.43 Mbps. The trade-off for this sizeable reduction in attack volume is a modest increase in bitrate. When the attacker is bandwidth-bound, its padded SYN floods contain approximately 11% more bits/second than its unpadded ones. Therefore SYN Padding is *only* effective when the primary threat comes from high *volume* rather than high *rate* attacks. That is, when the fixed overhead of handling each packet outweighs the per-bit overhead of handling larger packets.

6.3.2 Client Overhead and QoS Impact

Our SYN Padding mitigation is clearly effective at reducing the volume of SYN floods from bandwidth-constrained attackers, but we must evaluate its efficacy and overhead from the perspective of actual clients as well. To that end we conduct further experiments on the topology shown in Figure 6.1. This time the attack is launched from attackers `a4-a7`, on the far side of the bottleneck link. We test SYN Padding with the full 40 bytes of padding. We also measure the efficacy and overhead of SYN Cookies in the same context. Though not designed to mitigate high volume floods they provide a valuable point of comparison. We also test the effects of operating both mitigations simultaneously (referred to in our figures as "SYN Cookies + Padding"). Bottleneck link capacities of 1000, 100, and 50 Mbps are evaluated, with a 1ms bottleneck link latency in each case.

Our analysis begins by measuring clients' baseline performance, with results depicted in Figure 6.10. We should expect this to look essentially identical to the baseline measurements for SYN Cookies shown in Figure 6.3 since the choice of mitigation has no bearing on the baseline metric. However, our testbed is a complex dynamic system that cannot be perfectly controlled – if we run SYN Cookies experiments one day and SYN Padding experiments the next, we're likely to see subtle differences in *all* measurements, including the baseline. For this reason we repeat the full set of UB, MB, UA, and MA experiments for *every* set of variables we test, and we include both versions of the baseline data to illustrate how it can change over time. The key patterns are exactly the same as before however, with significantly better performance for clients located closer to the server and for applications that require fewer round trips to complete each transaction.

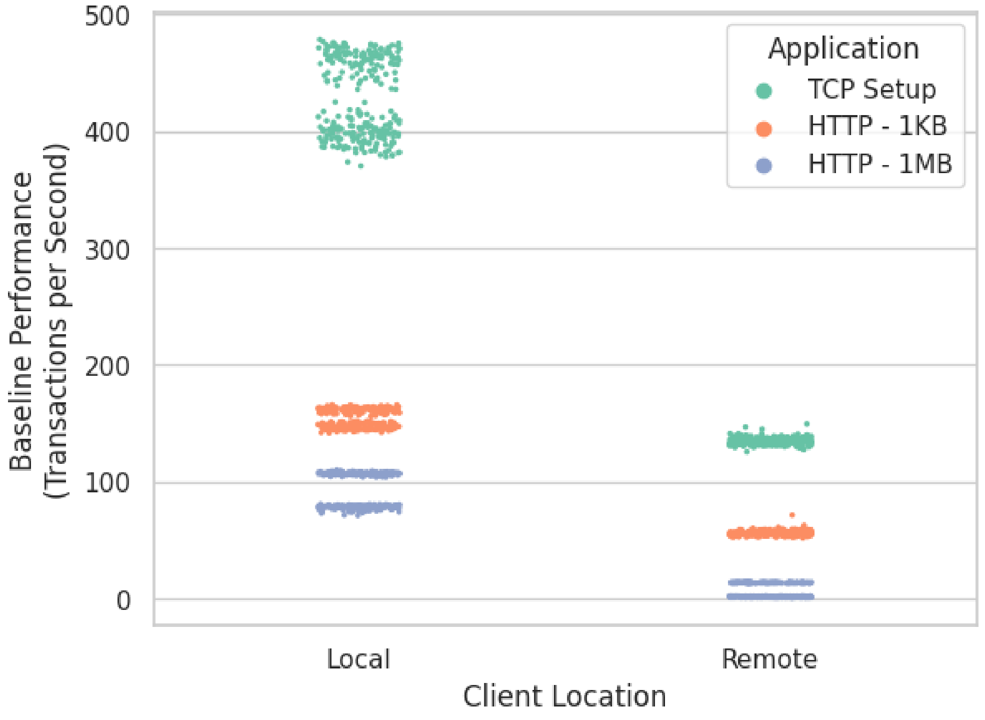

Figure 6.10: Baseline performance across client types, without the presence of an attack or mitigation. Local clients are positioned on the near-side of the bottleneck link to the server, remote clients are positioned on the far side. As expected, we observe significantly better performance for local clients, and for simpler transactions that require fewer packet exchanges to complete.

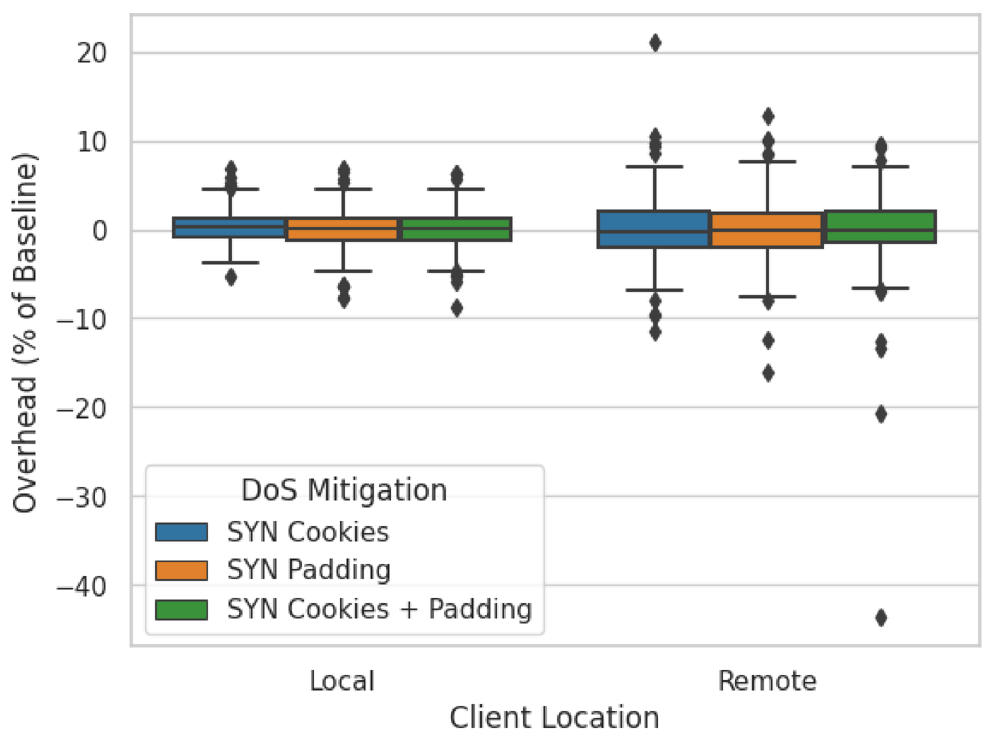

Figure 6.11: Overhead of our SYN-Padding mitigation compared with SYN-Cookies and the two combined. In all mean overhead is extremely near zero, with 95th percentile confidence interval within $+/-10\%$.

Next we evaluate the overhead of mitigation on client performance outside periods of attack. Figure 6.11 shows that the mean overhead is near zero for SYN Padding, SYN Cookies, and the two combined. We observe some outliers of both high overhead and extreme *negative* overhead (indicating that the mitigation somehow *improves* performance beyond the baseline). As with SYN Cookies, these can be accounted for by the variance observed in the baseline performance. From the application-layer perspective, the average overhead is negligible.

Next we introduce the attack, and as seen in Figure 6.12 the threat it poses is near-absolute. Like the baseline, threat is not a mitigation-specific metric, and so we see similar results as in Figure 6.5. The two HTTP clients lose 100% of their baseline performance in all

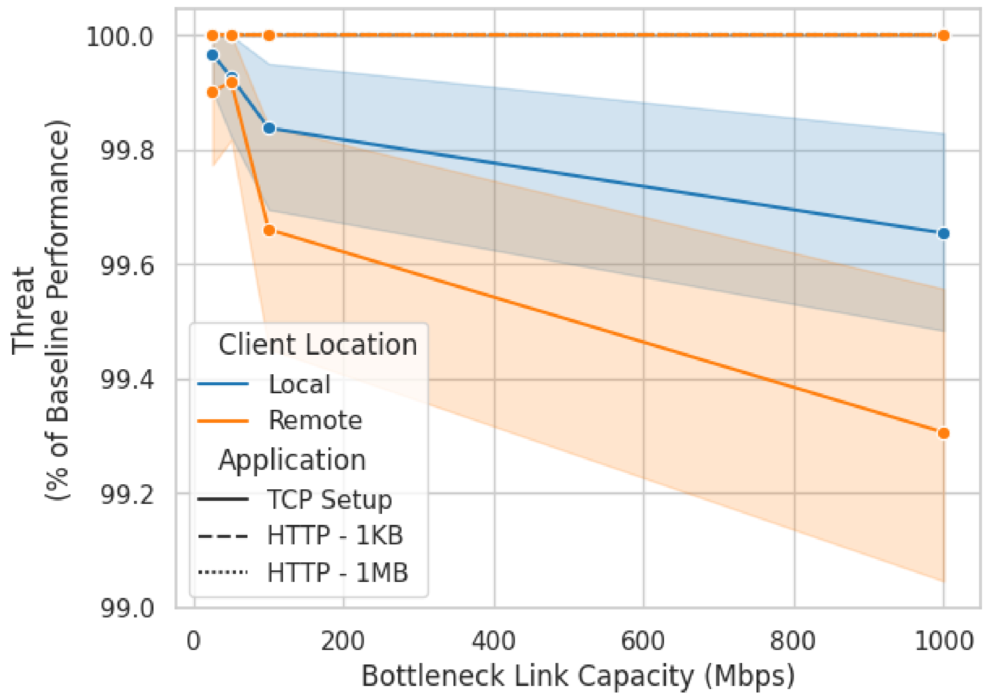

Figure 6.12: The threat posed by the attack. Lines for all HTTP client applications are overlapping at 100%, obscuring most of them. This includes local and remote clients attempting both 1KB and 1MB HTTP transfers. The simpler TCP clients manage to squeeze through a few occasional transactions, with slightly less threat to remote clients (because they start from a lower baseline) and under a less constrained bottleneck link (because fewer packets are dropped). These differences are faint, with the threat above 99% of in all contexts, but they do match our expectations. With a weaker attack and reduced threat we would expect to see the same differences magnified.

cases, and the TCP Setup client is reduced to less than 1% of its typical performance, with a slightly lower threat faced by remote clients (because their baseline is lower) and when the bottleneck is wider.

Finally we evaluate the efficacy of SYN Padding against both “smart” (padded) and “dumb” (unpadded) floods. We cannot hope to enforce particular behavior from attackers, only to incentivize it. We assume malicious actors will use any means available to disrupt communication as significantly as possible, without regard for standards or expectations. As such, attackers are free to choose whether to participate in the mitigation and pad their packets, or to send unpadded floods as usual. While we might expect them to prefer non-participation, failing to comply allows verifiers to drop *all* attack traffic early, whereas padded floods will still succeed in reaching their destination but at a reduced volume.

Results are depicted in Figures 6.13 (smart) and 6.14 (dumb). For smart floods we expect to see SYN Padding improve client QoS, because attackers are sending fewer, longer packets (as shown in §6.3.1). Indeed, we observe near 100% efficacy for clients local to the server, but near 0% for remote clients. In other words, the mitigation succeeds in reducing the attack volume enough to protect clients with very low latency, but the attack's threat is so severe that remote clients simply cannot compete with attackers. If the padding length were increased further and/or the flood rate were decreased, we would expect to see positive efficacy for remote clients as well. The location of the verifier process does not have a significant bearing on these results, as the padded attack packets will all pass verification regardless of location. In a smart flood, the mitigation's primary goal is already achieved the moment attack traffic is generated.

Dumb floods exhibit very similar trends, with one notable exception: when the verifier is located at the network edge, we achieve 100% efficacy for remote clients as well as for local ones. In this scenario the verifier is able to drop the *entire* flood at its first hop, sparing the rest of the network from congestion. This clearly illustrates the importance of verifier location, as well as the benefits of moving the verifier away from bottlenecks near the server and out towards attackers. It also demonstrates that attackers are incentivized to comply with the mitigation and pad their attack packets, since that allows them to realize the greatest damage, but either way SYN Padding succeeds in fully nullifying the attack's threat to local clients.

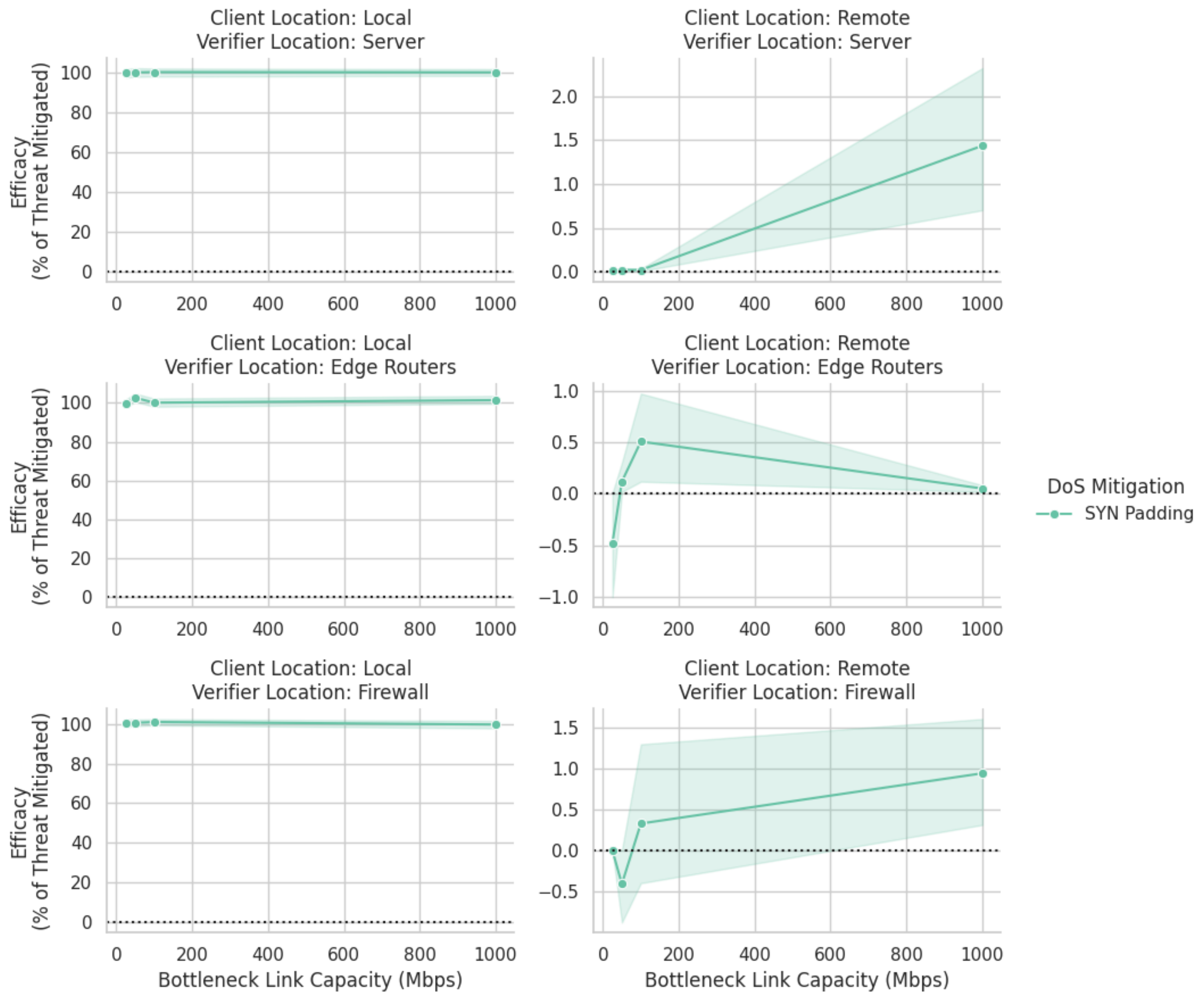

Figure 6.13: This figure shows efficacy of SYN Padding against a smart (padded) SYN Flood as a percentage of the threat mitigated. The left-hand column shows results for local clients (all near 100%) and the right-hand shows results for remote clients (all near 0%). Different rows show different verifier locations, though we do not observe a significant difference between them in this context. Bottleneck link capacity is shown on the x-axis but also makes no significant difference here. For local clients the attack is weak enough that we're able to fully mitigate the threat, yet for remote clients it is so strong that they can barely complete a single transaction. An extra network hop and a couple of milliseconds of extra latency makes an extraordinary difference.

One might assume a dumb flood is unrealistic in this context, since non-padded attack packets are guaranteed to be dropped as soon as they reach a verifier (unlike in SYN PoW where the dumb attacker will occasionally get lucky and stumble on a high-value hash). However, if we assume the primary resource bottleneck in the network relates to receiving and forwarding packets, then a flood of many small non-compliant packets could still be more damaging than one of larger standard packets. Even if the bad packets can be consistently detected and dropped, they still consume valuable resources en-route to the verifier. This is clearly evidenced by the poor efficacy for remote clients that we observe when the verifier is located at the server or firewall – it is still able to drop the entire flood, but not until after legitimate packets are dropped due to congestion on the bottleneck link.

Note that SYN Cookies also exhibit near 100% efficacy for local clients and near 0% for remote ones. However, unlike SYN Padding (and our general TIPs approach), SYN Cookies can *only* be deployed at the server itself – there is no flexibility to offload them to a firewall or edge router. This means that by the time attack traffic reaches the point at which SYN Cookies take effect, it has already transited (and likely congested) any bottleneck links on its path. This highlights a crucial advantage of our approach: that it enables much earlier drops of malicious traffic, potentially as early as the first hop. Combining the two mitigations produces no notable synergies (positive or negative), with performance tracking closely to that of SYN Padding alone. Adding SYN Cookies does not improve performance beyond the equal or superior efficacy achieved by SYN Padding, nor does either mitigation detract from the other.

In summary, we find SYN Padding has negligible overhead from the client perspective, and that it provides extremely high efficacy for certain clients, primarily those close to

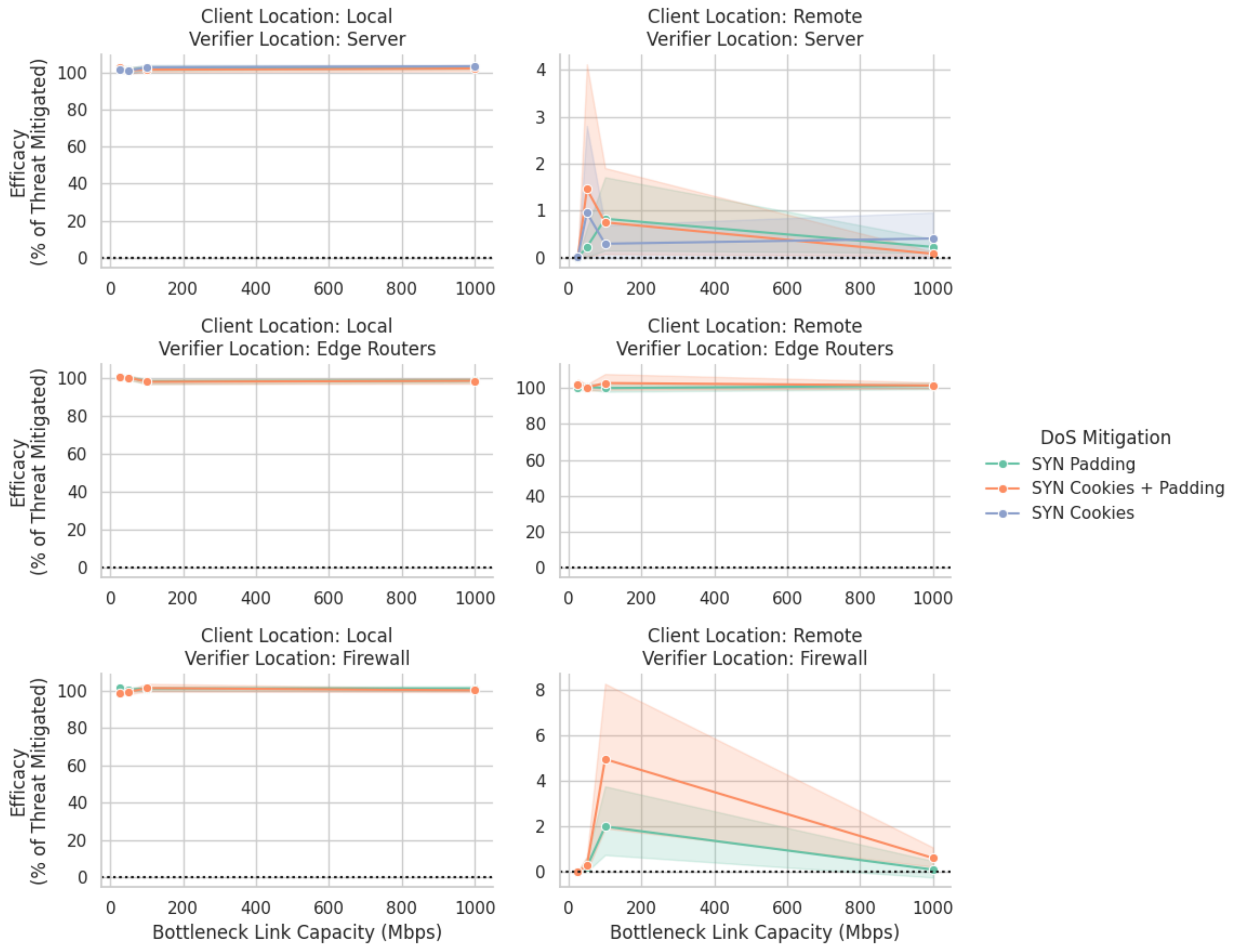

Figure 6.14: This figure shows efficacy of SYN Padding against a dumb (unpadded) SYN Flood as a percentage of the threat mitigated. We also compare against SYN Cookies, and test the two mitigations simultaneously. The left-hand column shows results for local clients (all near 100%) and the right-hand shows results for remote clients, where we see an interesting divergence from the smart attack in Figure 6.13. Efficacy is again near zero when the verifier is deployed at the server or its first-hop firewall router (r0), but looking at the middle row we see it jump up to 100% when the verifier is deployed at the edge routers (r1-r3). In this case the verifiers are able to drop the *entire* flood one hop away from its source, before it is able to exhaust critical network resources. This clearly illustrates the importance of verifier location, and the value of moving verification away from targeted devices towards the attack source. Again we see that client location is the single most important factor however, even obscuring effects of constraining the bottleneck link down to 50Mbps – even that weakened attack is sufficient to take remote clients offline. Note that we only show data for SYN Cookies alone when the verifier is located at the server (top row) since they cannot be implemented elsewhere. Performance of the two mitigations combined is at least equivalent to that of SYN Padding alone, and for remote clients with verification at edge routers SYN Padding vastly outperforms SYN Cookies.

the server. When verifiers are pushed to the network edge, near the attack source, efficacy increases for remote clients as well, in some cases providing 100% efficacy where SYN Cookies provide near 0%.

6.4 SYN PoW

We now present the evaluation of our SYN PoW mitigation, defined in Section §4.2.3. Following the same structure as our analysis of SYN Padding above, we first explore the mitigation's objective ability to reduce the volume of a SYN flood in Section §6.4.1, followed by its subjective overhead and efficacy from the client perspective in Section §6.4.2.

6.4.1 Traffic Rate/Volume Reduction

We begin our analysis of SYN PoW by measuring the amount by which it is able to reduce the volume of traffic produced by a single attacker. It is important to observe that the feature we are varying on the x-axis here is the average number of iterations of the hash function, rather than the bottleneck bandwidth as in our similar analysis of SYN Padding above. This represents the burden being put onto the CPU capacity rather than the network bandwidth capacity. This burden is per-packet rather than per-bit.

Again, these experiments are performed on a subset of the topology depicted in Figure 6.1. For each set of parameters tested, a 30 second SYN flood is launched from node a0 with a destination of node s0. The SYN PoW verifier is deployed on the ingress side of node r0's network interfaces, with varying proof thresholds. Node s0 records the average attack **rate** in bits per second and the average attack **volume** in packets per second, using the `tcpstat` utility. This tool outputs one data point for every 5 second window – we exclude the first and last data points for each trial as their windows include time before the attack starts and after it ends. We then average results from the middle 20 seconds which represent the real steady-state nature of the attack.

Results are shown in Figure 6.15. We first measure the maximum attack rate a0 is able to produce, with no verifier running and no proof being performed. It achieves a steady-state attack rate of 164.1 Mbps (446.0 Kpps), enough to pose a significant threat to many real-world systems. We next deploy the verifier at node r0, sequentially testing values of $k \in \{8, 32, 64, 128, 256\}$, but still without any proof-of-work being performed at the attacker. In these "dumb" floods, only $\frac{1}{k}$ of all attack packets generated will have sufficient hash values, thereby allowing the verifier to drop the remaining $\frac{k-1}{k}$. The dashed black line in Figure 6.15 indicates the expected attack rate, assuming hash values are truly random and the verifier performs perfectly, defined by the function $y = \frac{max(y)}{x}$. Our experimental results match this theory precisely, as is plainly visible in the overlap between the blue and dashed-black lines. We also provide the raw data in Table 6.2, including a comparison of attack rate and volume along with the expected and actual percent reductions in each.

With the highest k value measured of 256, we succeed in reducing the attack to just 0.38% of its original rate, from 164.1 Mbps (445.97 Kpps) down to just 0.63 Mbps (1.71 Kpps). We have every reason to expect that even higher proof thresholds would continue to follow the theoretical model and produce even smaller attacks. There are diminishing returns to increasing k however, as clearly illustrated by the sharp elbow in Figure 6.15. There are also increasing costs to legitimate clients, as higher k values add more latency to their connection setup.

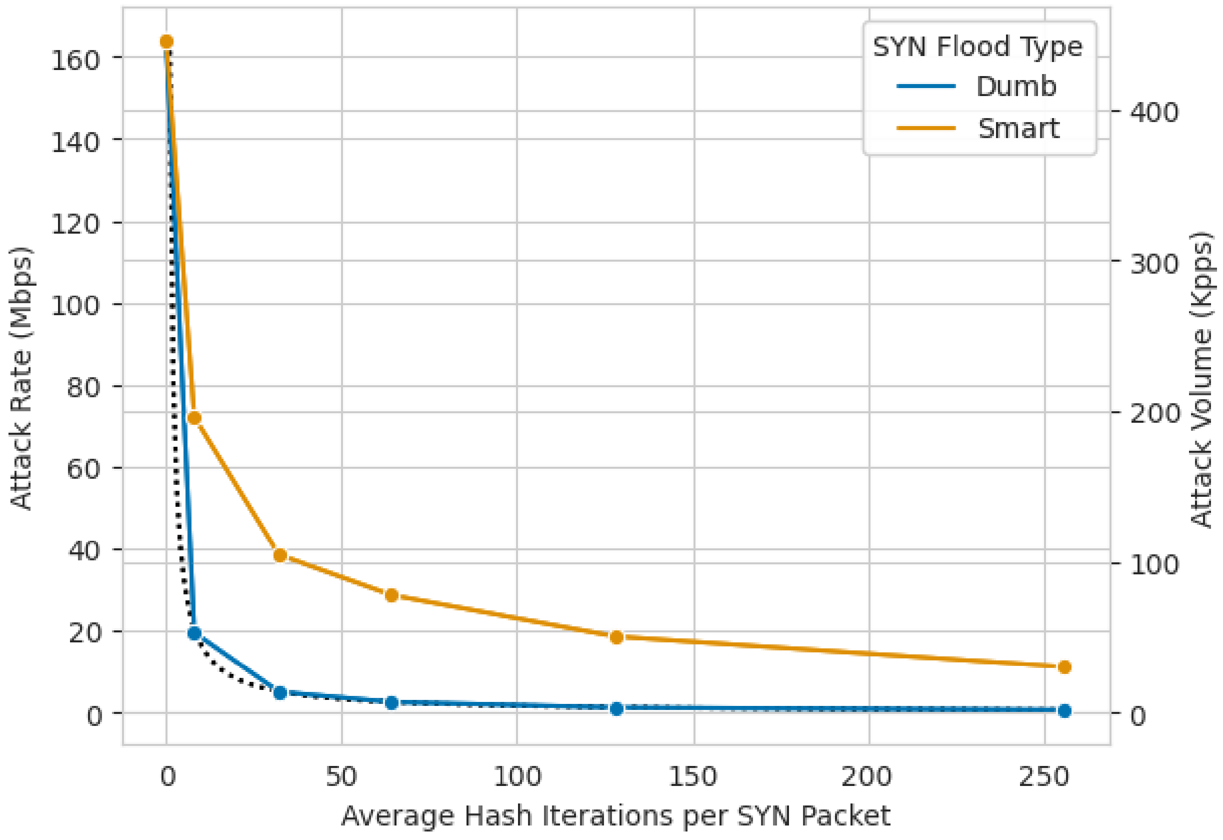

Figure 6.15: The rate-limiting effect of our SYN PoW mitigation against a single volumetric SYN flooder, based on the expected number of hash iterations per packet, k. We compare "smart" floods in which the attacker performs the proof against traditional dumb floods in which they send SYNs as quickly as possible. The dashed line represents the equation $y = \frac{MaximumAttackRate}{k}$, the optimal rate reduction we can expect to achieve against a dumb flood in which $\frac{k-1}{k}$ of attack packets are dropped. We use a dual y-axis to indicate both the attack's bit-rate and its packet-rate, which are directly proportional to one another due to the fixed size of attack packets.

k	Rate (Mbps)		Volume (Kpps)		% of Maximum Rate		
	Theory	Real	Theory	Real	Theory	Real	Difference
8	20.51	19.53	55.75	53.07	12.5	11.90	0.60
32	5.13	5.05	13.94	13.72	3.13	3.08	0.05
64	2.56	2.55	6.97	6.93	1.56	1.55	0.01
128	1.28	1.22	3.48	3.31	0.78	0.74	0.04
256	0.64	0.63	1.74	1.71	0.39	0.38	0.01

Table 6.2: A comparison of our experimental results with the theoretical model for SYN PoW's ability to reduce the rate/volume of "dumb" SYN floods. As expected, the verifier is able to drop almost exactly $\frac{k-1}{k}$ of all attack packets. The rightmost column group indicates the mitigated attack rate as a percentage of the maximum (unmitigated) attack rate of 164.1 Mbps. We succeed in reducing the attack to a small fraction of its original volume, with efficacy that matches expectation precisely.

While SYN PoW is clearly extremely effective at reducing the threat of traditional SYN Flooding attacks, we must also consider a "smart" attacker that attempts to perform the proof-of-work operation honestly for each of its SYNs in order to get past the verifier. We repeat the same procedures as for the "dumb" floods, this time with the SYN PoW prover installed on node a0 while it launches the attack, using the same k value as the verifier to ensure all generated packets are accepted. These results are shown with the orange lines in Figure 6.15. We see that smart attackers do pose a slightly greater threat than dumb ones, but the added burden of computing proofs is still highly effective at rate-limiting them. In the best case with $k = 256$, we succeed in reducing the attack rate to 11.13 Mbps (30.26 Kpps), which is just 6.78% of the attacker's normal capability. Even with the trivially low threshold of $k = 8$ the smart attack is reduced to 72.06 Mpbs (195.83 Kpps), less than half its original size.

Next we restrict the attacker's packet generation process to use only 10% of the system's available CPU resources using the `cpulimit` utility. Results are presented in Figure 6.16. With no mitigation in place this CPU throttling reduces the maximum attack rate to 17.16

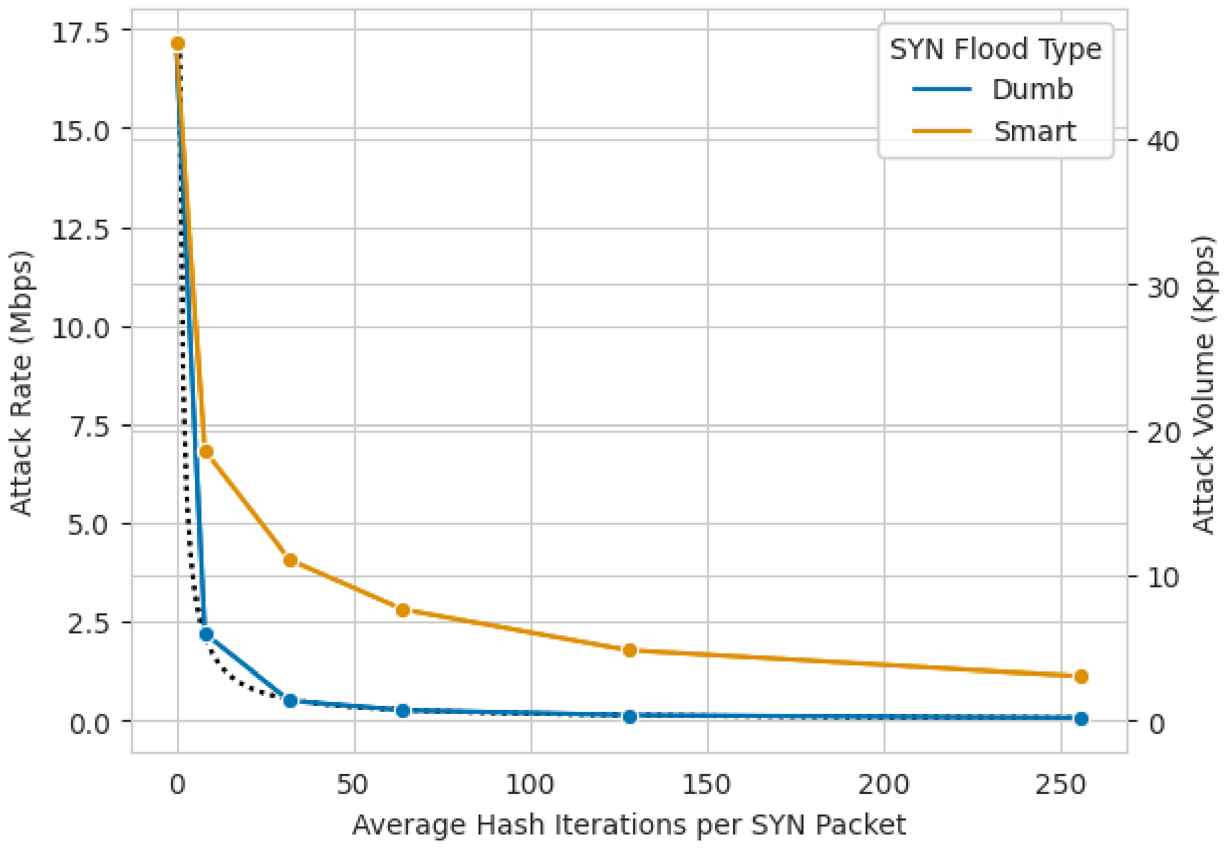

Figure 6.16: This figure mirrors Figure 6.15, but with the attacker's packet generation process restricted to 10% of its available CPU resources. The result shows almost identical trends at $\frac{1}{10}^{th}$ scale, indicating that our attacker was already CPU-bound, and that the SYN-PoW mitigation scales well with changes in attacker resources. It doesn't matter how severe the attacker's CPU bottleneck is, only that it is their primary bottleneck. We again succeed in dropping almost exactly $\frac{k-1}{k}$ of all packets in the "dumb" SYN floods, and up to 93.5% of packets in "smart" floods (when $k = 256$).

Mbps (46.64 Kpps), which is 10.46% of the maximum rate attainable when CPU is unrestricted. This indicates that our attacker was already CPU-bound, as expected. SYN PoW remains effective at restricting the flood further however, with a nearly identical trend as in Figure 6.15 for both dumb and smart SYN floods. Against dumb floods we again succeed in dropping almost exactly $\frac{k-1}{k}$ packets for all thresholds tested, as illustrated by the overlap of the blue and dashed-black lines. With the maximum threshold tested of $k = 256$ the flood drops to just 1.12 Mbps (3.03 Kpps) – 6.5% of the CPU-limited maximum rate.

6.4.2 Client Overhead and QoS Impact

Our SYN PoW mitigation is clearly effective at reducing the volume of SYN floods, whether attackers attempt to perform proofs or not. This is encouraging as we generally expect the threat posed by volumetric DoS attacks to scale with their size, but it is not yet sufficient to recommend widespread deployment. We must also evaluate how much overhead legitimate clients incur in performing proofs, and what the mitigation's actual net impact is on application-layer QoS.

To that end we conduct experiments on the full topology shown in Figure 6.1. We measure two variants of SYN PoW, with k=8 and k=64. We also compare against the special k=0 case in which no proofs are performed or validated, in order to measure the overhead incurred by simply loading an eBPF program. We label this as the "eBPF No-Op" mitigation. Again, we also test SYN Cookies (discussed in §3.2) as a point of comparison.[5] We first launch the flood from just the four attackers closest to the server, a0-a3, and then with the full set of eight to compare attacks of different strengths. We also restrict attackers with the `cpulimit` utility as discussed above, testing limits of 100, 50, 10, 5, and 1 percent of available CPU. For each version of the SYN PoW mitigation tested, we measure both "smart" and "dumb" flood types; in smart floods attackers compute proofs for each packet and in dumb floods

[5]Unfortunately we were unable to test the effects of SYN PoW and SYN Cookies simultaneously. There is no inherent conflict between the two in terms of design, but there does appear to be one between the specific implementations we use. Our eBPF-based SYN PoW implementation alters the SYN's Acknowledgement Number field to encode its nonce, while SYN Cookies encode information in the SYN-ACK and ACK, based on contents of the SYN. The two *should* be fully isolated from one another, but when attempting to combine them our clients fail to establish any connections. We have yet to identify the precise cause of this conflict, but hypothesize that SYN Cookies are *assuming* a value of zero for the Acknowledgement Number field in one part of their operation and using the actual value (which is *usually* zero) in another, causing a mismatch that leads to the rejection of legitimate client ACKs. We remain optimistic that the two mitigations can co-exist, and note that this ambiguous behavior of SYN Cookies is emblematic of the problems that can arise when protocol modifications are deployed without sufficient standardization.

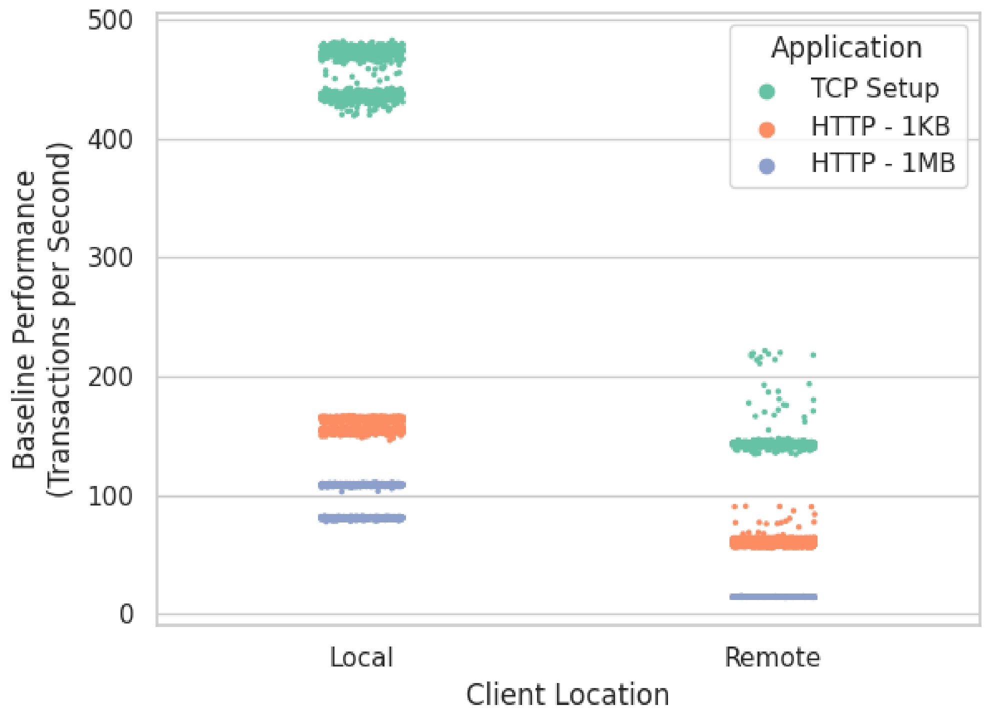

Figure 6.17: Baseline performance across client types, without the presence of an attack or mitigation. Local clients are positioned on the near-side of the bottleneck link to the server, remote clients are positioned on the far side. As expected, we observe significantly better performance for local clients, and for simpler transactions that require fewer packet exchanges to complete.

they do not.

In all cases, we have the full set of twelve clients attempt to communicate with the server, using the same three client applications tested with SYN Cookies and SYN Padding above (TCP Setup, HTTP 1KB, and HTTP 1MB). Figure 6.17 illustrates how baseline performance differs across these applications, and how it depends on client location. As expected, results are nearly identical to those depicted in Figures 6.3 and 6.10, since baseline performance is in no way mitigation-specific. Again we see local clients outperform remote ones, and simpler applications outperform those that require more round trips.

Next we evaluate the overhead of each mitigation on client performance outside periods

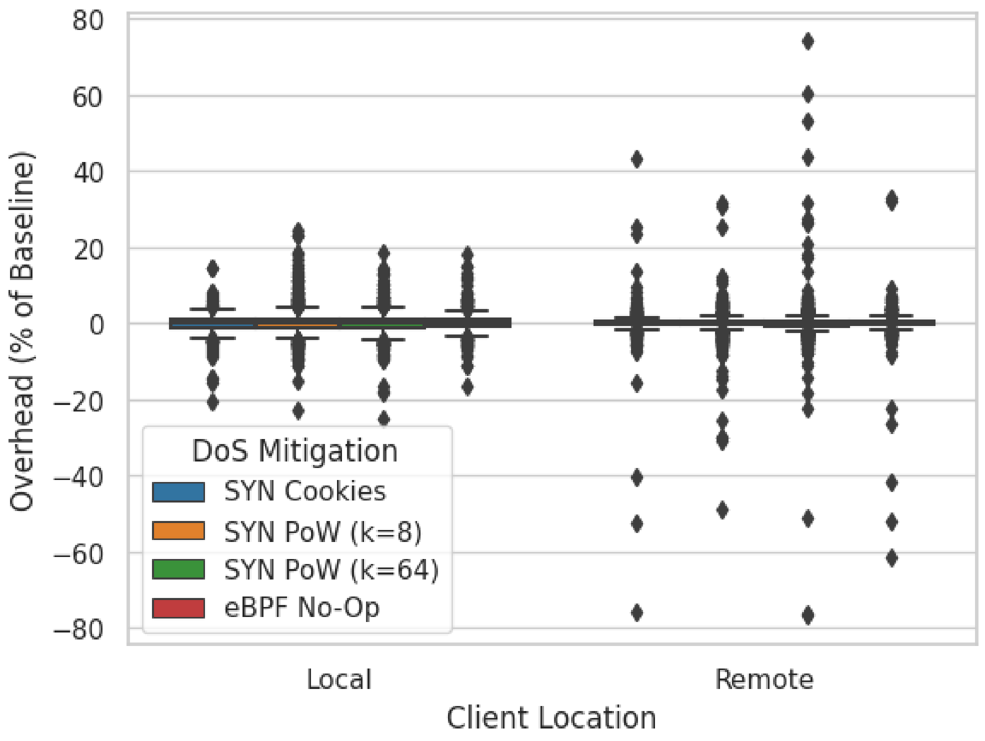

Figure 6.18: Overhead of our SYN PoW mitigation compared with SYN-Cookies and our eBPF No-Op program. In all cases mean overhead is extremely near zero, with 95th percentile confidence interval extending to +/- 5%. Bars are so narrow that the colors are difficult to see, but their left-to-right order matches the top-to-bottom order in the legend.

of attack. Figure 6.18 shows that this overhead is similarly low for all four mitigations, with no discernible difference between the overhead of a No-Op eBPF program and one that actually performs a proof, nor between the k=8 and k=64 variants. We again observe some outliers of both high overhead and extreme *negative* overhead (indicating that the mitigation somehow improves baseline performance), both of which can again be accounted for by the variance observed in the baseline. Remote clients experience significantly higher variance in both baseline and overhead than local ones do. In all cases the mean is near zero and the confidence interval extends equally above and below the x-axis, indicating that overhead for all four mitigations is truly negligible from the application-layer perspective, relative to natural fluctuations in baseline performance.

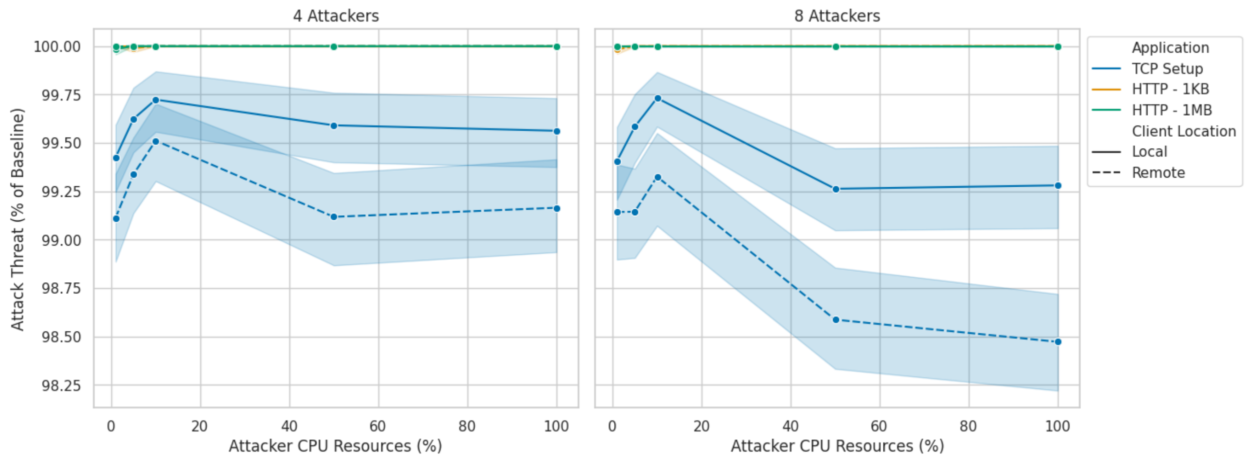

Figure 6.19: The threat experienced by clients during the attack as a percentage of the Baseline. We see that the threat is absolute for HTTP clients with all lines overlapping at 100%, and is consistently above 98% for the simpler TCP Setup.

Introducing the attack, first with four devices and then eight, we see that the threat is again near-absolute. Figure 6.19 depicts this threat as a percentage of the baseline performance. For the two HTTP clients the threat is a full 100%, and the TCP Setup client's performance is reduced by over 98% in all cases. Local clients still slightly outpace remote ones, but due to their large advantage in baseline performance they also perceive a slightly greater threat.

Finally we evaluate the efficacy of our mitigations against the flood. Figure 6.20 depicts efficacy for each as a percentage of the baseline, split across local/remote clients, smart/dumb floods, and 4/8 attackers. Starting with the standard "dumb" flood, we again we see a large rift between local and remote clients, with 80-100% of the threat successfully mitigated for all local clients and remote clients struggling near 0% efficacy for all but the weakest floods. Interestingly the performance of our our SYN PoW mitigations tracks very closely to that of SYN Cookies regardless of the attack's strength. If attackers attempt the proofs and send "smart" floods, clients observe even better efficacy (though as we will see this does not necessarily mean attackers are incentivized to continue sending dumb floods). Even the remote clients observe near 100% efficacy in the 4-attacker smart flood, as well as significant improvements with 8 CPU-constrained attackers. Here we see the clearest difference between the k=8 and k=64 variants, with k=64 creating a drastic reduction in attack rate that restores client QoS to 100% of the baseline in all cases except remote clients with 8 attackers. SYN Cookies on the other hand never exceed 30% efficacy for remote clients (as seen in the 4-attacker dumb flood graph). Note that our experiments included additional variables (most notably the client application and verifier location), but the narrow error bands (indicating the 95th percent confidence interval) illustrate that these have little impact on the results. Attack volume and client location are by far the dominating factors in determining mitigation efficacy.

In summary, our SYN PoW mitigations exhibit negligible overhead and consistently outperform SYN Cookies at mitigating volumetric SYN Floods, with the higher-difficulty k=64 variant offering marginally better protection than k=8. These results make a compelling case for widespread deployment of SYN PoW on the Internet, though we emphasize that further experimentation and standardization through the IETF or a similar body must be performed first.

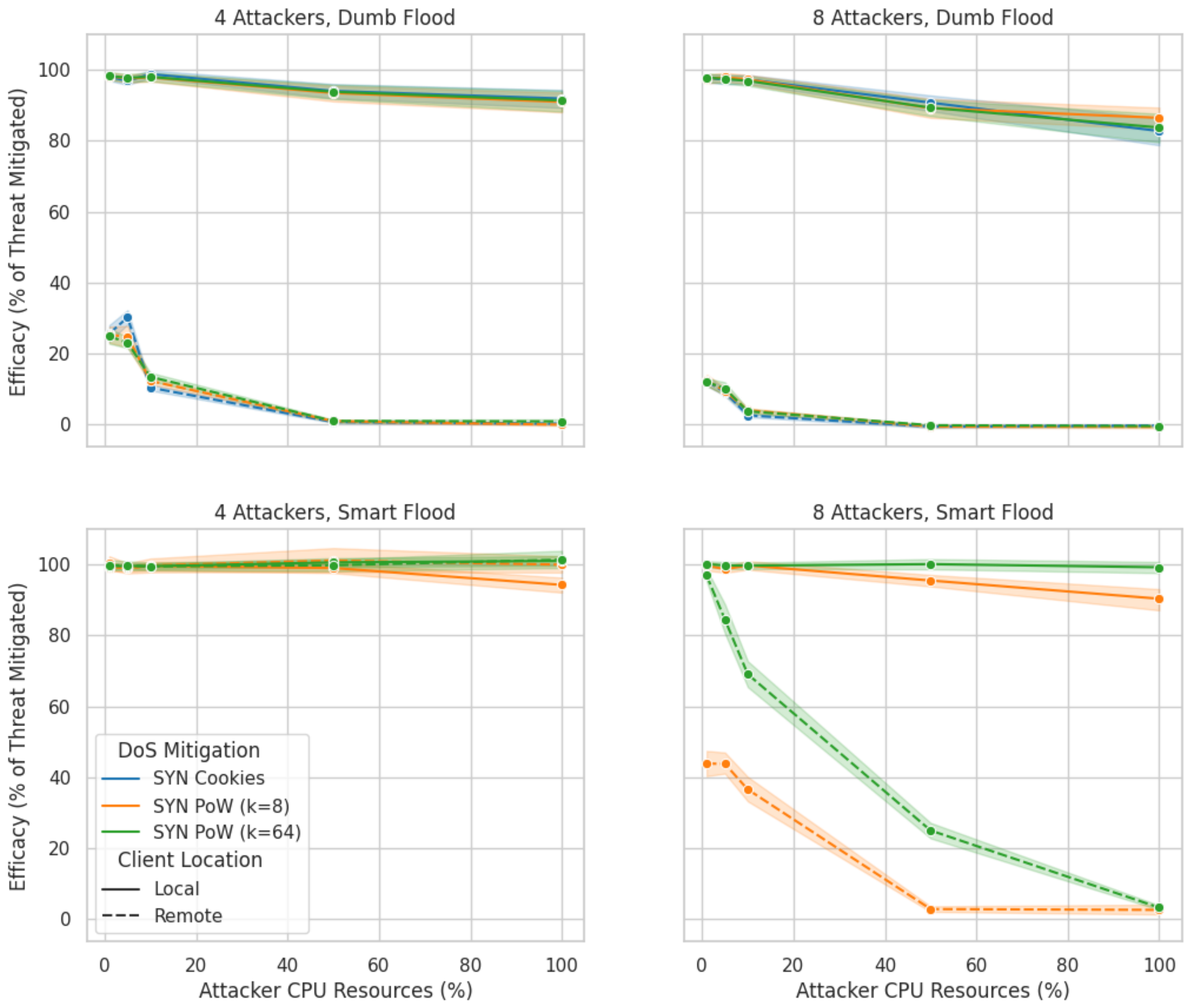

Figure 6.20: The Efficacy of our SYN PoW mitigations, compared with that of SYN Cookies, for local and remote clients across varying flood types. Local clients (in subnets A and B) achieve extremely high efficacy in all cases, often nearing 100% and only dipping to near 80% when faced with the full strength of a dumb flood from 8 attackers. For remote clients the threat of the dumb flood is simply too severe to salvage any performance except when attackers are limited to below 50% of their normal CPU resources. With the smart flood however, they acheive near 100% efficacy against 4 attackers. In the 8-attacker smart flood we see the most significant impact of attacker CPU limitations and the clearest difference between k values. As the CPU limit decreases, efficacy moves from 0% up to 50% for $k = 8$ and all the way up to 100% for $k = 64$, verifying that higher proof thresholds can yield higher efficacy. Note that with sufficiently strong attack we would eventually see efficacy drop to zero for local clients, and with weaker attacks we would see it increase to 100% for remote clients. Finding the precise effective range for either group would require extensive trial and error. We can say that in this context the mitigations are highly effective for local clients and occasionally effective for remote ones.

Chapter 7

Discussion

In this chapter we discuss potential future work in this domain and extensions to our contributions, as well as limitations of this work and other miscellaneous considerations. In Section §7.1 we discuss how the TIPs mitigation approach (described in Chapter §4) might be implemented in protocols other than TCP. Considerations for supporting both our TCP implementations and others in IPv6 are presented in Section §7.2. Section 7.3 presents Hash-Sorted Queuing, an extension to our SYN-PoW mitigation (defined in §4.2.3) which provides multi-tiered service based on the amount of work clients perform rather than verifiers making a binary accept/drop decision. In Section §7.4 we explore how our measurement framework (presented in Chapter §5) can be used to compare the DoS resilience of disparate network protocols and internet architectures. Section §7.5 discusses the limitations of that framework and additional metrics that may be useful. Section §7.6 discusses challenges of generating consistent high-volume floods for experimentation, and Section §7.7 outlines our data collection, storage, and analysis pipeline. Finally, we discuss additional considerations for deploying the novel mitigations we've implemented in Section §7.8, before briefly summarizing in §7.9.

7.1 Non-TCP Implementations

We chose to implement the first proofs-of-concept for our TIPs approach to mitigation in TCP because TCP SYN floods pose a severe and immediate threat. They are far from the only threat however, and the general concept of bootstrapping trust in a single packet via provable resource expenditure is one that can and should be applied to other protocols. There are far too many potential avenues for deployment to enumerate, but we present likely candidates for both padding- and PoW-based mitigations below.

7.1.1 Padding

As shown in §6.3, packet padding can provide a valuable tool for forcing bandwidth-constrained attackers to send fewer, longer packets. Assuming the predominant overhead of an attack is per-packet rather than per-bit, this simple intervention can have a drastic impact on performance. A similar implementation in UDP may also be valuable, as that represents one of the two largest vectors for short-packet volumetric attacks alongside SYN floods. However, the UDP header lacks the option fields we use for padding in TCP, meaning we would need some form of application-layer support to ensure that padding in the data payload is properly ignored by the receiver. UDP also lacks TCP's flags to distinguish special packet types – SYNs represent only a small fraction of all TCP packets sent by ordinary clients so padding them adds little overhead, but in UDP the only option is to pad *all* packets. Deployment in QUIC may be valuable as well – it has been rapidly gaining market share at the transport layer and its initial packets resemble TCP SYNs, most notably in their rarity of use and ease of identification. QUIC does not presently present a significant DoS threat, but we strongly believe in the importance of proactive rather than reactive approaches to mitigation. While padding can be effective at the transport layer, it is likely better positioned at the network layer. This is because the per-packet resources that short packets consume primarily exist at that layer and below. The real goal of packet padding is to protect the network's most essential functionality of forwarding packets, regardless of their contents.

Determining the optimal minimum packet size, assuming we are unburdened by restrictions of existing standards, is a challenging task. Forcing attackers to send longer packets can reduce the damage caused by volumetric floods, but forcing ordinary clients to pad their packets would cause a consistent reduction in network efficiency. However, there is an argument to be made that any "legitimate" clients which send extremely short packets are already operating in a way that reduces efficiency. Naively, we expect that including more data in each packet means fewer forwarding resources are required to deliver the same amount of information, though the reality is a bit more complex. Larger packets are more expensive to re-transmit, making them inefficient on networks with a high loss rate. Additionally, any given device has only a finite amount of data to send at any given time – increasing *minimum* packet lengths above the *average* packet length sent by legitimate clients would be guaranteed to impose overhead.

The ideal packet size therefore depends on some combination of attackers' outgoing bandwidth, average packet sizes, and typical Internet-wide loss rates. Unfortunately we are unaware of sufficient data sources for determining any of these factors, let alone all three. While we strongly suspect the current minimum packet size of 20 bytes in IPv4 is inefficiently small, we cannot recommend a specific alternative without further research. We also acknowledge that altering such a key aspect of such a ubiquitous protocol is a near impossibility on the modern Internet, and urge those developing future internet architectures to consider the potential DoS implications of packet length in their designs.

7.1.2 Proof-of-Work

As discussed in earlier chapters, our Proof-of-Work approach to DoS mitigation is far more robust than packet padding, and we expect it to be applicable across a wide range of protocols. The two key requirements are that attackers be CPU bound (which we assume to be the norm), and that there exists some natural asymmetry between attackers and legitimate clients. For SYN PoW the primary asymmetry we exploit is that attackers send SYN packets in much higher volumes than legitimate clients do, meaning attackers take the brunt of the cost added by the mitigation. We expect similar imbalances exist within other packet types used in volumetric attacks. In fact, that imbalance is largely what *defines* a volumetric attack, when devices send more than their fair share of traffic.

In the context of short-packet UDP floods (which are effectively IP floods), the primary imbalance is packet length. Attackers send very short packets far more often than normal clients do – indeed, a PoW could be included in UDP (or IP) as an alternative to padding. By setting proof thresholds inversely proportional to packet length, shorter packets would require more CPU resources to generate than longer ones. This sort of hybrid approach could offer benefits of both padding and PoW, at the cost of additional complexity. It would protect receivers' bandwidth resources from CPU-bound attackers, but would require a combination of both their verifiers as well – we would expect to see additive effects in efficacy, but also in overhead. There is no clear place to add a nonce to the UDP header, and adding one to the payload would require higher-layer support as with padding. There are no convenient fields available in the IPv4 or IPv6 headers, but a PoW nonce could be added as a new IPv4 option or a new IPv6 extension header. In designing new protocols a dedicated nonce field could easily be included. It may also be possible to combine the PoW hash with a network- or transport-layer checksum in order to reduce overhead. The same hash function may not be optimal for both however, and any benefits to efficacy would not justify a compromise in security.

PoW-based mitigation could also be useful in HTTP(S), (S)FTP, QUIC, or other data transfer applications, with difficulty set corresponding to the value/scarcity of the resource being requested. Files that are larger, replicated across fewer devices, or located farther from the requester could require higher-value proofs than small, common, or close files. In this case the assumed asymmetry is that ordinary clients only request such high-value files on rare occasion. If a file is widely replicated that is generally *because* it is commonly requested by clients, and if a real user regularly needs the same very large file they are generally better off downloading it once and keeping a local copy. Devices that *repeatedly* behave outside these norms are likely to be malicious, but we don't want to risk over-penalizing devices that only *occasionally* send odd/challenging requests. As with padding (and with SYN PoW), there is a delicate balance to strike in determining how much work provers should be required to put in. Appropriate thresholds should be determined through experimentation, following the model provided in Chapters §5 and §6. For comparison, existing application-layer mitigations may add multiple seconds of latency or even require human interaction (as discussed in §3.2.3) – we would expect to see high efficacy from PoW-based solutions with overhead on the order of a millisecond per packet or less.

7.2 IPv6 Support

One immediate next step for this work is to provide IPv6 support in our implementations of SYN Padding and SYN PoW. This will add a small amount of complexity to the verifier programs, since IPv4 and IPv6 headers require sepearte code to parse, but we do not expect this will add significant overhead. We do not anticipate any other hurdles to IPv6 deployment of these mitigations.

It's worth noting that in IPv6 a single device can *legitimately* own and use a large number of addresses simultaneously, since address space is so abundant. This means IPv6-based flooders may be even more difficult to detect than those in IPv4, further motivating implementation of our mitigations in that architecture.

7.3 Hash-Sorted Queuing

One possible extension to PoW-based mitigation is to replace the binary accept/drop threshold with a mechanism we call **Hash-Sorted Queuing** (HSQ). As the name suggests, packets are instead sorted into different queues based on their hash value. Packets bearing extremely high-valued proofs may be rewarded with priority service while lower values incur more delay, and only the lowest-valued packets will be dropped when necessary. Figure 7.1 illustrates this concept.

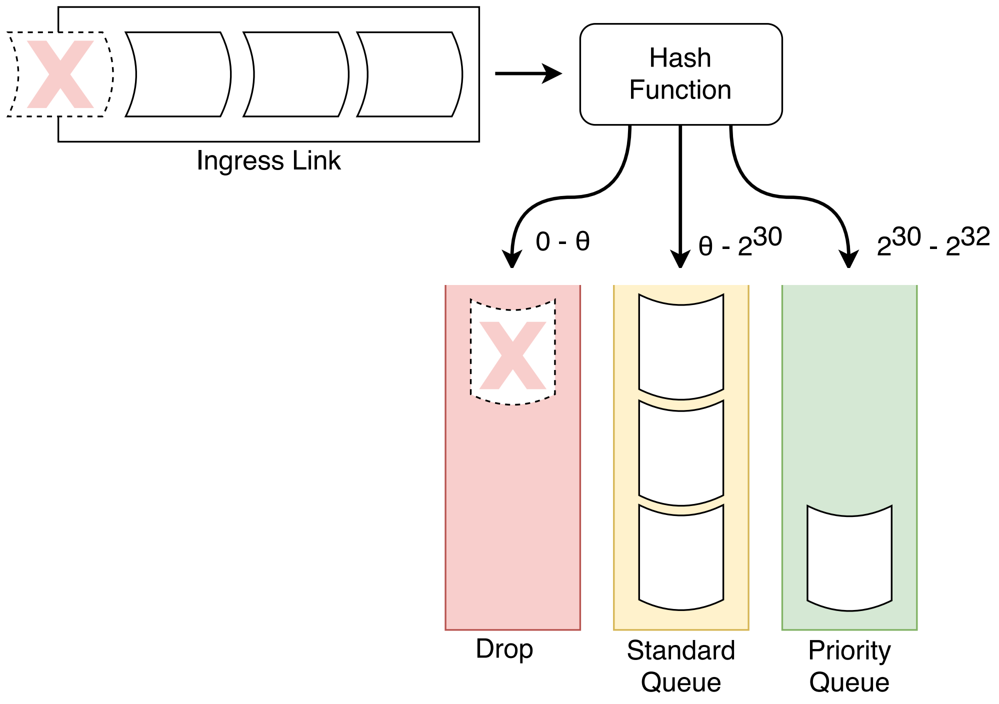

Figure 7.1: An illustration of our proposed Hash-Sorted Queuing mechanism. The verifier computes the hash value of each incoming packet and assigns them to different queues accordingly. Packets with high-valued hashes are given priority while those with the lowest values may be dropped. The verifier can wait until the standard queue becomes full to begin dropping, or it may choose to take a more proactive approach and drop any packet with a hash value below the threshold θ regardless of queue length. During periods of extreme demand, some packets may need to be dropped before they reach the verifier. We assume the hash output is 32 bits in length, allowing for hash values ranging from $0 - 2^{32}$.

As with ECC (described in §4.2.3.5), clients could use AIMD or a similar algorithm to determine how much effort they should expend on proof generation in order to receive adequate QoS, or servers could announce thresholds through the DNS. Again, those methods add complexity, but removing the binary accept/drop decision also removes some drawbacks of variable-threshold approaches because the consequences of guessing the threshold incorrectly are less severe. If a client guesses low and performs less work than expected, their packet will be delayed rather than dropped (unless congestion is extremely high or their proof value is extremely low). If they guess high and perform extra work, they will be rewarded with superior service quality rather than that work being wasted.

Ultimately we envision this creating a sort of free market for packets, in which devices eventually reach an equilibrium at which senders “pay” just enough to receive whatever quality of service they desire. If a service provider sets unreasonably high thresholds, clients would be incentivized to seek an alternative, sparking competition. In theory this could even provide a mechanism for enforcing the peering agreements on which BGP relies – rather than policing traffic volumes from neighboring ASes, border gateways could simply set different proof thresholds for each AS in a way that prioritizes traffic to/from their higher-paying customers.

There is one major hurdle facing HSQ that is shared by all proposals for providing differentiated services: establishing consensus on what constitutes “good” service. As discussed in §3.3, different applications can have drastically different resource requirements. Clients seeking to minimize latency want their packets sent as quickly as possible, while those seeking to minimize loss may prefer their packets be held temporarily if that means they will be sent over a more reliable link. If adding PoWs to an application-layer protocol it may be possible for the verifier to infer the prover’s desire, assuming all users of that application have similar requirements. If implemented at lower layers, verifiers should not examine higher-layer headers or payloads to make a similar inference, as that sort of deep packet inspection can compromise network neutrality. There may be value in adding proofs at multiple layers, but each one should be handled purely within its own layer. Therefore, low-level verifiers should prioritize every packet they receive uniformly (in most cases by minimizing latency). Alternatively, a few additional bits could be added to each packet, signifying the sender’s desired prioritization method from a standard list of options.

7.4 Protocol and Architecture Comparison

The original impetus for this research was an attempt to evaluate the DoS-resilience of proposed future internet architectures [84], in comparison against one another and against the current TCP/IP model. In starting down that path we quickly realized that we lacked any robust method for quantifying DoS-resilience, including that of the current architecture. This ultimately led to development of the measurement framework presented in Chapter §5, which is designed to facilitate exactly this sort of analysis. Our metrics and methodologies can be used to compare IPv4 against IPv6, and against a novel architectures like Named Data Networking (NDN) [85], [86] or MobilityFirst [87]. They can also be used to compare protocols at any layer, allowing us to answer: whether QUIC is really a suitable replacement for TCP; how DNS over HTTPS (DoH) compares to traditional DNS over UDP; or whether a given application provides better end-user QoS when running on TCP or UDP. We can also compare different implementations of the same protocol, to establish which operating system is most efficient, or to tune parameters of a complex mitigation towards optimal values.

Even with this framework, DoS-resilience (and conversely, DoS-vulnerability) is a highly subjective trait. It depends on particulars of the attack launched, the services under attack, the relative locations of participants, the software and hardware they use, etc.. We can state whether one architecture, protocol, or implementation is superior to any other, but only within a specific context. We can also begin to look at how altering that context alters the comparison – perhaps IPv6 outperforms IPv4 for certain applications, but only when the RTT or hop-count is below some threshold. Through further experimentation we hope to identify common trends in these cross-context measurements that will offer deeper insights into *why* some architectures and protocols are more resilient than others, which can then inform more objectively DoS-resilient designs in the future.

7.5 Additional Metrics

This section discusses limitations of our measurement framework, defined in §5, and additional measurements that may be beneficial. We discuss economic costs (§7.5.1), time-series analysis (§7.5.2), and estimations of attack probability (§7.5.3).

7.5.1 Economic Cost Modeling

In §3.2.2 we mentioned that mitigating volumetric DoS attacks via over-provisioning, while effective, imposes high monetary costs. Purchasing more devices than are needed to handle legitimate demand is wastefully expensive. While our measurement framework is primarily designed to optimize performance, it can be adapted to evaluate this economic overhead as well. A simple mitigation in this case is to increase the number of servers in our network from one to two. The cost is plainly obvious: it is the price of acquiring, deploying, and maintaining that second server. There may still be some marginal performance overhead as well, for example if the new server is positioned farther away from clients or is equipped with different hardware resources than the first. Even the seemingly simple act of load balancing traffic between two servers adds some overhead to the system, and potentially the addition of yet another device on which to perform that load balancing (though it may be done in advance via the DNS). In measuring the efficacy of adding a second server, it is useful to test how many legitimate clients can be supported per device, as well as how strong of a flooding attack they can withstand.

Of course this is only one piece of a much more complex equation – service providers that rely on over-provisioning must also consider the potential costs of *not* acquiring excess capacity. If they start dropping legitimate packets, either as the result of a volumetric attack or a legitimate spike in demand, their customers will be upset and they may lose business. For certain mitigations, latency is so severe that it imposes measurable economic costs. As discussed in §3.2.3, the average user takes 32 seconds to solve (re)CAPTCHA puzzles, which totals an estimated 500 years per day of wasted time. If we value that time at the meager United States federal minimum wage of $7.25/hr, it represents *$11.59 billion per year* worth of lost productivity.

Finding the right balance between efficacy and overhead can be an extremely challenging task, and often it is better to be safe than sorry – to *guarantee* service persists during extreme floods, even if that means lower performance and/or higher costs during normal operation. However, we believe the costs of existing mitigations must be examined more closely. Anecdotally, we have heard multiple colleagues in the field express a belief that CDN-based mitigation services have "solved" the problem of volumetric DoS attacks, when in reality they are converting performance costs to economic ones. This is a tremendously valuable ability to have in our arsenal of defense strategies, but it is not infinitely valuable – sometimes the best option may be for service to fail (temporarily). Without a complete understanding of what DoS mitigations cost, we cannot make informed decisions about when and where they are worth deploying.

7.5.2 Time-Series Analysis

Our metrics all aggregate data from multiple transactions, over extended periods of time. We have found this approach necessary in producing useful data visualizations – our experiments involve a large number of variables, and the more we aggregate our data the more variables can be displayed simultaneously. However, this aggregation may also hide interesting features of the data we collect. Prior work in measuring DoS attacks and mitigations has made frequent use of time-series analysis, wherein some performance metric is plotted on the y-axis of a scatter plot or line graph, with time on the x-axis. This allows us to visualize changes in performance over time, most notably the ramp-up and ramp-down periods at the start and end of the attack. How long does it take between the time the attack starts and the time QoS reaches some new, reduced steady state? How long between the end of the attack and a return to the original steady state? Are those two steady states actually very steady, or is there a high variance in QoS (with or without an attack or mitigation)? Time-series plots can be very helpful in answering these questions. In our experiments we've found systems may take upwards of 30 seconds to return to normal after a flood ends, as the TCP server continues re-transmitting responses to attack packets. The ramp-up phase at the start of an attack is near-instantaneous however. Variance during the steady state depends largely on the client application and features of the network topology. In summary, time-series analysis can still provide a valuable supplement to the metrics we have presented, though it makes multi-variate cross-context analysis difficult.

7.5.3 Estimating Attack Probabilities

Some of the metrics presented in §5.5 require information about what volume of attack a device *expects* to receive with what frequency. If high-volume attacks are common, then it may be worth deploying a high-overhead mitigation if it's able to provide high efficacy. If attacks are rare, the mitigation's efficacy becomes less valuable and its overhead is harder to justify. Finding the right balance of efficacy and overhead is crucial to optimizing performance, yet predicting attack patterns is a difficult task. To be clear, devices do not need to predict if/when any specific attack event will occur, only the probability distribution of traffic volumes they will receive over time. Still, we are unaware of any existing data sources that could be used to make this determination in the general case, and while individual receivers may be able to predict how much *total* traffic they will receive with reasonable accuracy, determining what *portion* of that traffic belongs to an attack is a separate challenge – if attack packets were easily identifiable they would simply be dropped. Until additional data can be collected, network and device operators are forced to guess how often the mitigations they deploy will actually be useful, and how often they are serving as dead weight. Fortunately the mitigations we tested exhibit negligible overhead from the client's perspective, so deploying them is likely to have a net benefit.

7.6 Attack Traffic Generation

Accurately evaluating DoS mitigations requires us to generate DoS attacks – we flood devices under our own control and measure how they perform. To test high-volume floods, we either need a large number of devices (as in a real-world botnet), or a way to send large amounts of traffic from a single device. Since resources in our testbed are limited, our only choice is the latter. To that end we have developed a highly efficient packet generation script in the C programming language. The fastest approach we have found is to allocate a single buffer which is re-used for each packet in the flood. Before sending the next packet we generate a new random source IP address (skipping private and reserved address blocks), and modify *only* the bits in our packet buffer corresponding to that field. We then re-compute the TCP and IP checksums and modify bits for those fields as well before sending the new packet. This avoids any unnecessary copying between buffers, and we have found it can produce consistently faster and more consistent floods than reading pre-generated packets from memory.

Note that it is important that we are able to generate packets manually, with full control over every field at every layer. DoS experimentation requires us to test corner cases, to check whether a potential new attack vector could be damaging. This is cannot be done by simply replaying packet captures of past real-world attack incidents, since some major threats have yet to be realized.

Our flooding scripts are made available along with the rest of our research code at [3], though we stress that they are to be used for research purposes only, never to launch attacks against any real-world devices. We urge caution even to the most well-intentioned reader: experiments should always be performed in a securely sandboxed environment. Even targeting one's own devices can cause harmful effects to intermediary nodes and recipients of backscatter traffic, if proper precautions are not taken.

7.7 Data Analysis Pipeline

After running a suite of experiments we transfer the collected data from the DeterLab testbed to CSAIL's OpenStack cluster, where there are more abundant resources for storage and analysis. Our data analysis pipeline is fully automated, including the processing of raw data to extract our various metrics, and the generation of figures from those metrics. We parse data in python, using Jupyter notebooks to enable automation, ensure consistency, and foster reproducibility.

Processed data is stored in a PostgreSQL database using a custom schema we have developed to mirror the DeterLab workflow. This is depicted in Figure 7.2, which was generated automatically from our schema file by the service https://dbdiagram.io. This schema provides the flexibility to add arbitrary new parameters to our experiments (by storing them as JSON objects) while still enabling efficient queries. We generate figures directly from database queries using the `seaborn` and `matplotlib` python libraries. Our own prior work provides a more in-depth discussion on designing database systems for DoS mitigation measurement research [88].

Development of these data processing tools required a significant amount of time and effort, and while they lack the direct intellectual merit of this thesis's primary contributions, we hope others will find them useful in conducting experiments similar to ours (or in attempting to repeat our work). These tools are provided along with the rest of our research code at [3].

7.8 Deployment Considerations

When discussing modifications to existing protocol standards, we must be mindful to design mitigations that are not only technically sound but also practically feasible to deploy, from the perspectives of economics and public policy. Ideally this requires approaches that can be deployed incrementally. Protocol modifications may take years to standardize and potentially decades to actually realize widespread deployment, so we cannot expect to make any sweeping changes to the whole Internet architecture overnight.

A prime example is the transition from IPv4 to IPv6. In February 2011, ICANN announced the final allocation of IPv4 address space, describing the event as “major turning point in the on-going development of the Internet,” yet acknowledging that "no one was caught off guard," because "the Internet technical community [had] been planning for IPv4 depletion for some time." Indeed, the first version of the IPv6 standard was published over 16 years prior in December 1995 [89]. Yet global adoption remains below 50% to this day [90].

To avoid such delays we should design mitigations that are incrementally deployable, able to co-exist with legacy devices that have not yet implemented them, and straightforward to deploy on a wide variety of hardware and software. We have striven to meet this standard in the design of our proposed SYN Padding and SYN PoW mitigations, most notably by implementing them as portable eBPF programs. Since eBPF allows for arbitrary packet analysis and modification, we see this as a promising approach for deploying other mitigations in the future. Meaningful change will not happen overnight on the Internet, but with careful and clever design we can help it happen faster.

7.9 Summary

In summary, our immediate next steps to build on this work are to conduct further experiments to fine-tune our SYN Padding and SYN Pow mitigations, and to provide IPv6 support for them. Looking slightly longer-term we plan to develop in-kernel implementations of both mitigations; to develop implementations of similar padding- and PoW-based approaches for other protocols facing pressing DoS vulnerabilities; and to extend our PoW model to realize the vision of Hash-Sorted Queuing in which individual packets directly “pay” (in CPU cycles) for the services they solicit. We also plan to begin comparative analyses of potential future internet architectures, and encourage others to do the same using our measurement framework.

We discussed limitations of our metrics and methodologies, including how they might be adapted to capture monetary costs in addition to changes in performance; how aggregating over extended periods of time may hide some interesting features of the data we collect; and why some of the metrics we define *cannot* reasonably be measured unless we are able to obtain better data describing expected attack volumes.

Additionally, we describe the automated pipeline used to analyze our data; discuss generating large volumes of attack traffic in a research setting; and consider strategies for designing mitigations that can be deployed easily and incrementally, in order to facilitate faster adoption.

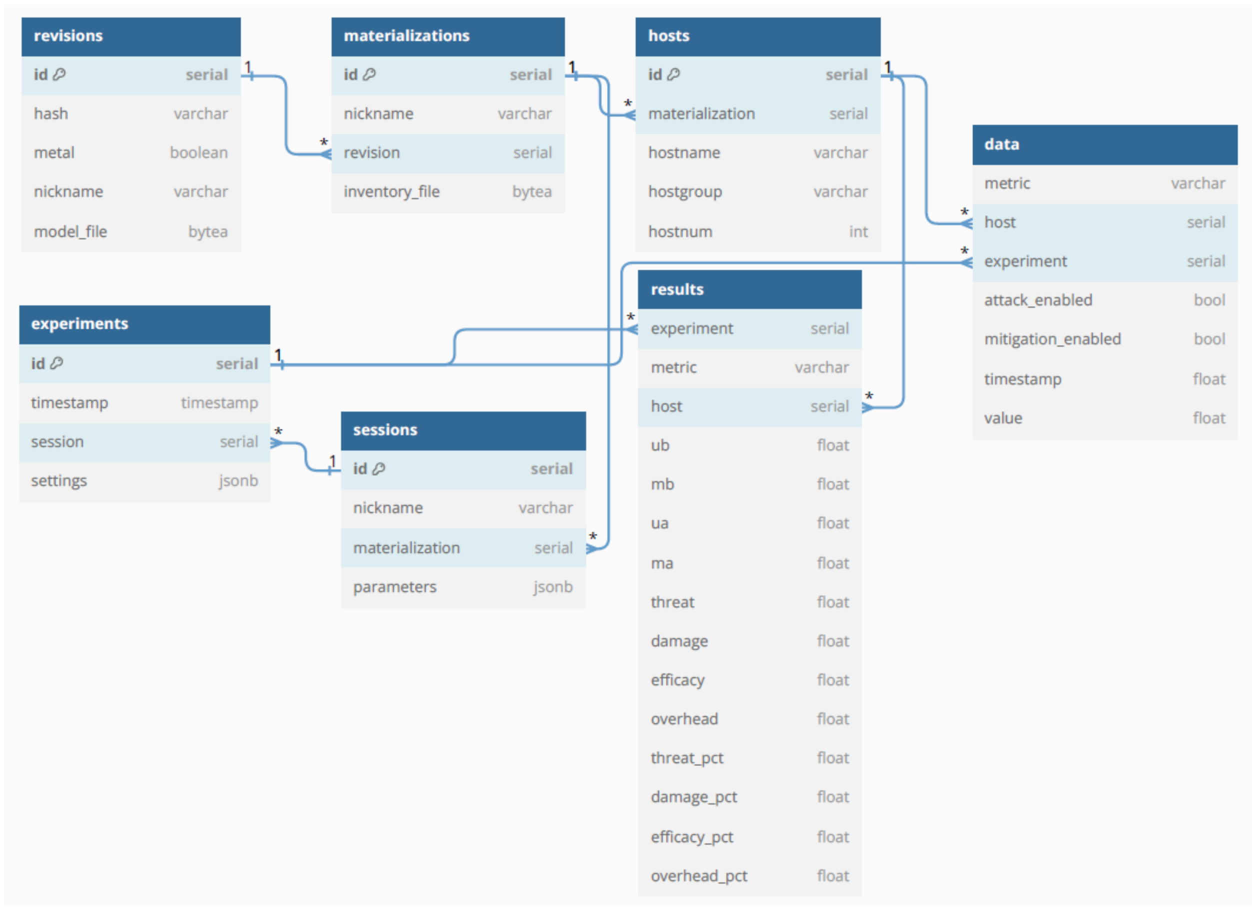

Figure 7.2: The database schema used to organize results of our experiments, designed to meld the structure of our measurement framework with that of the DeterLab testbed on which our experiments are conducted. Experimentation in DeterLab starts by creating a **revision**: an abstract definition of a network topology. From that revision we can then create a **materialization**: a manifestation of the topology on a specific set of hardware in the testbed. Each materialization has some set of **hosts**, which may be a combination of physical devices and virtual machines. Our measurement framework operates in units of **experiments**, each of which includes a set of the four core measurements (UB, MB, UA, and MA) described in §5.4. Typically we conduct multiple experiments in a single **session**, systematically testing all possible permutations from a given set of parameters. Each experiment results in a large number of **data** points, which are summarized into **results**.

Chapter 8

Conclusion

Volumetric Denial-of-Service attacks pose a severe and ever-growing threat to the Internet. They exploit inherent insecurities in its architectural design: devices must handle traffic from other devices with which they are unfamiliar, and *generating* a large volume of traffic is much easier than *receiving* the same volume. Existing mitigations have significant costs, which are predominantly borne by the very devices they aim to protect. This thesis has outlined a path towards designing a fundamentally more DoS-resilient Internet, by increasing the costs of flood generation and making it easier to detect and drop bad packets closer to their source.

We re-frame volumetric DoS mitigation as a performance optimization problem, wherein the goal is to shift bottlenecks away from attack targets towards the network edge and onto attackers themselves. We observe that significant asymmetries exist between attackers and regular clients, which allow us to influence the behavior of attackers without needing to identify them or their traffic directly. Making certain packets marginally more difficult to generate can create a drastic reduction in flood volume with negligible overhead.

We show that simple packet padding can be an effective mitigation against certain attacks, serving as verifiable proof that a sender has expended significant bandwidth resources. Bandwidth-bound attackers that pad their packets will send fewer, longer packets per second. In volumetric attacks, we assume that the benefits of receiving fewer packets outweigh the costs of receiving longer ones. Attackers that choose to continue sending unpadded floods can maintain their normal rate, but once their traffic reaches a verifier it can easily be dropped. Our implementation of this concept in TCP, SYN Padding, was largely designed as a toy example yet provides extremely high efficacy in certain contexts. While we do not recommend deploying this mitigation as-is, it suggests a clear link between packet length and DoS resilience. We believe that mandating a longer minimum packet length at the network layer could provide valuable protection against high-volume short-packet floods.

Using a hash-based proof-of-work system provides an even more effective, flexible, and scalable version of this concept. We expect the vast majority of attackers are CPU-bound, and so requiring them to expend just a few extra CPU cycles per packet is an easy way to reduce their flood volume. For regular clients this extra work amounts to small fraction of a round-trip time and is unnoticeable at ordinary sending rates. Again, if attackers choose not to participate and send packets without proofs, those packets can easily be dropped. In this case verification is a bit more complex than checking the packet length, as verifiers must compute one iteration of the hash function themselves, but it still has negligible impact on client QoS. Proofs are also probabilistic, meaning non-participating attackers will occasionally get lucky and *appear* as though they are participating, but a single attacker may send hundreds of thousands of packets per second so this variance will quickly average out. The system can easily be tuned to make the attacker's job arbitrarily difficult, at the expense of slight increases in client latency. Since attackers send orders of magnitude more packets per second, they experience orders of magnitude more impact from the added difficulty. We have provided a standards-compliant design for SYN PoW, an implementation of this concept designed to combat the ubiquitous TCP SYN flood, and implemented both the prover and verifier as eBPF programs to maximize efficiency and portability. Further experimentation, fine-tuning, and standardization will be required to prepare SYN PoW for widespread deployment, but we believe it has the potential to radically transform the way we approach volumetric DoS mitigation.

Both these mitigations, SYN Padding and SYN PoW, follow our general concept of Trustworthy Independent Packets (TIPs) – they bootstrap trust between unfamiliar devices in a single packet, without relying on prior state or additional communication. Crucially, this allows the role of verification to be decoupled from the packet's destination. It can be performed at a nearby firewall to offload processing from a resource-constrained server, or distributed across edge devices in order to detect and drop bad traffic (with insufficient proofs) closer to its source. This gives us a crucial advantage over existing mitigations like SYN Cookies, which require direct involvement from the packet's destination and therefore can *only* respond to attacks once significant resources have already been spent forwarding the flood through the network. Moreover, SYN Cookies and most similar mitigations *actively respond* to attack traffic, further increasing strain on the network and potentially harming secondary targets with backscatter traffic. With TIPs, responses are sent only if verification succeeds. Indeed, our experimental results clearly show that SYN Cookies steadily lose efficacy as attack volume increases, and that using our TIPs approach with a verifier placed near the attack source can overcome this to provide exceptional protection during an ordinarily devastating flood.

To evaluate our proposed mitigations, we have defined a suite of metrics and a rigorous experimental methodology. Controlling for both the separate and combined effects of an attack and a mitigation on a system allows us to tease apart a mitigation's efficacy from its overhead, enabling more accurate assessment of its net utility. We show that small changes in network topology or device behavior can lead to drastically different results, reinforcing the importance of measuring mitigations across a wide variety of realistic contexts. This measurement framework can be adapted to track arbitrary metrics – we rely primarily on client-side application-layer indicators of performance, but we discuss how economic costs can be measured as well. We can also compare the DoS-resilience of any two protocols, architectures, implementations, topologies, etc. using this model, by simply framing a change from one to the other as a mitigation. Our code and other data is made available to facilitate further research in this domain [3].

In summary, we urge a closer analysis of the costs existing DoS mitigations impose, and propose novel approaches based on packet padding and proof-of work which can help redistribute those costs away from attack targets and towards their sources.

Bibliography

[1] C. Martinho and T. Strickx, *Understanding How Facebook Disappeared from the Internet*, http://blog.cloudflare.com/october-2021-facebook-outage/, Oct. 2021. (visited on 02/03/2022).

[2] S. Gibbs, “Google reinforces undersea cables after shark bites,” *The Guardian*, Aug. 2014, ISSN: 0261-3077. (visited on 07/21/2023).

[3] S. DeLaughter, *Research Data for Doctoral Thesis: Redistributing the Costs of Volumetric Denial-of-Service Mitigation*, https://samd.is/phd. (visited on 08/28/2023).

[4] G. E. Moore, “Cramming More Components onto Integrated Circuits,” *Electronics*, vol. 38, no. 8, pp. 114–117, Apr. 1965.

[5] G. E. Moore, “Cramming More Components Onto Integrated Circuits,” *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[6] N. Woolf, “DDoS attack that disrupted internet was largest of its kind in history, experts say,” *The Guardian*, Oct. 2016, ISSN: 0261-3077. (visited on 07/08/2022).

[7] D. Menscher, *Identifying and protecting against the largest DDoS attacks | Google Cloud Blog*, https://cloud.google.com/blog/products/identity-security/identifying-and-protecting-against-the-largest-ddos-attacks, Oct. 2020. (visited on 03/20/2023).

[8] A. Gutnikov, O. Kupreev, and Y. Shmelev, “DDoS Attacks in Q1 2022,” Kaspersky Securelink, White Paper Q12022, 2022. (visited on 04/29/2022).

[9] Azure Network Security Team, *2022 in review: DDoS attack trends and insights*, https://www.microsoft.com/en-us/security/blog/2023/02/21/2022-in-review-ddos-attack-trends-and-insights/, Feb. 2023. (visited on 03/19/2023).

[10] Cybersecurity and Infrastructure Security Agency (CISA), Federal Bureau of Investigation (FBI), and Multi-State Information Sharing & Analysis Center (MS-ISAC), *Understanding and Responding to Distributed Denial of Service Attacks*, Oct. 2022. (visited on 07/19/2023).

[11] Cybersecurity and Infrastructure Security Agency (CISA), *Capacity Enhancement Guide: Additional DDoS Guidance for Federal Agencies*, Oct. 2022. (visited on 07/19/2023).

[12] Common Criteria Recognition Arrangement, “Common Criteria for Information Technology Security Evaluation,” Common Criteria Recognition Arrangement, Tech. Rep. CC:2022 Release 1, Nov. 2022. (visited on 07/19/2023).

[13] M. Antonakakis, T. April, M. Bailey, *et al.*, "Understanding the Mirai Botnet," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1093–1110, ISBN: 978-1-931971-40-9. (visited on 02/03/2022).

[14] B. A. Karpf, "Dead reckoning : Where we stand on privacy and security controls for the Internet of Things," Thesis, Massachusetts Institute of Technology, 2017. (visited on 11/18/2017).

[15] B. Schneier, *The Internet of Things Is Wildly Insecure — And Often Unpatchable*, https://www.wired.com/2014/01/theres-no-good-way-to-patch-the-internet-of-things-and-thats-a-huge-problem/, Jan. 14. (visited on 12/14/2016).

[16] K. R. Sollins, "IoT Big Data Security and Privacy Versus Innovation," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 1628–1635, Apr. 2019, ISSN: 2327-4662. DOI: 10.1109/JIOT.2019.2898113.

[17] O. Yoachimik, *DDoS Attack Trends for 2022 Q1*, http://blog.cloudflare.com/ddos-attack-trends-for-2022-q1/, Apr. 2022. (visited on 10/05/2022).

[18] O. Yoachimik, *DDoS Attack Trends for 2022 Q2*, http://blog.cloudflare.com/ddos-attack-trends-for-2022-q2/, Jul. 2022. (visited on 10/05/2022).

[19] A. Gutnikov, O. Kupreev, and Y. Shmelev, "DDoS attacks in Q2 2022," Kaspersky Securelink, White Paper Q22022, Aug. 22. (visited on 10/06/2022).

[20] V. D. Gligor, "A Note on Denial-of-Service in Operating Systems," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 3, pp. 320–324, May 1984, ISSN: 1939-3520. DOI: 10.1109/TSE.1984.5010241.

[21] R. M. Needham, "Denial of service," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, 1993, pp. 151–153.

[22] J. Mirkovic and P. Reiher, "A Taxonomy of DDoS Attack and DDoS Defense Mechanisms," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 39–53, Apr. 2004, ISSN: 0146-4833. DOI: 10.1145/997150.997156. (visited on 12/13/2016).

[23] A. Kuzmanovic and E. W. Knightly, "Low-rate TCP-targeted Denial of Service Attacks: The Shrew vs. The Mice and Elephants," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '03, New York, NY, USA: ACM, 2003, pp. 75–86, ISBN: 978-1-58113-735-4. DOI: 10.1145/863955.863966. (visited on 11/16/2017).

[24] M. Luckie, R. Beverly, R. Koga, K. Keys, J. A. Kroll, and k. claffy, "Network Hygiene, Incentives, and Regulation: Deployment of Source Address Validation in the Internet," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, London United Kingdom: ACM, Nov. 2019, pp. 465–480, ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3354232. (visited on 09/20/2021).

[25] R. Beverly, A. Berger, Y. Hyun, and k. claffy, “Understanding the Efficacy of Deployed Internet Source Address Validation Filtering,” in *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC ’09, New York, NY, USA: ACM, 2009, pp. 356–369, ISBN: 978-1-60558-771-4. DOI: 10.1145/1644893.1644936. (visited on 11/17/2017).

[26] Anna-senpai, *Mirai Source Code*, https://github.com/jgamblin/Mirai-Source-Code, Oct. 2016. (visited on 12/13/2016).

[27] Cloudflare, *What is the Mirai Botnet?* https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/. (visited on 06/28/2023).

[28] I. Zeifman, D. Bekerman, and B. Herzberg, *Breaking Down Mirai: An IoT DDoS Botnet Analysis*, https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html, Oct. 2016. (visited on 12/13/2016).

[29] Internet Engineering Task Force, *RFC Index*, https://www.rfc-editor.org/rfc-index.html. (visited on 07/19/2023).

[30] Cloudflare, *What is DDoS mitigation?* https://www.cloudflare.com/learning/ddos/ddos-mitigation/. (visited on 06/09/2023).

[31] Apposite Technologies, *What is Deep Packet Inspection (DPI)?* https://www.apposite-tech.com/what-is-deep-packet-inspection-dpi/, Jul. 2021. (visited on 06/28/2023).

[32] R. Derby, *Detect, Prevent, Improve with Scalable DPI*, https://www.netscout.com/blog/detect-prevent-improve-scalable-dpi, Feb. 23. (visited on 06/28/2023).

[33] Fortinet, *What Is Deep Packet Inspection (DPI)?* https://www.fortinet.com/resources/cyberglossary/dpi-deep-packet-inspection. (visited on 06/28/2023).

[34] M. A. DeRose, “Deep Packet Inspection and its Effects On Net Neutrality,” Ph.D. dissertation, Regis University, 2010. (visited on 06/28/2023).

[35] N. Mehrabi, F. Morstatter, N. Saxena, K. Lerman, and A. Galstyan, “A Survey on Bias and Fairness in Machine Learning,” *ACM Computing Surveys*, vol. 54, no. 6, pp. 1–35, Jul. 2022, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3457607. (visited on 06/28/2023).

[36] F. K. Došilović, M. Brčić, and N. Hlupić, “Explainable artificial intelligence: A survey,” in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2018, pp. 0210–0215. DOI: 10.23919/MIPRO.2018.8400040.

[37] Damian Menscher, *Exponential Growth in DDoS Attack Volumes*, https://cloud.google.com/blog/products/identity-security/identifying-and-protecting-against-the-largest-ddos-attacks, Oct. 2020. (visited on 06/09/2023).

[38] W. M. Eddy, "TCP SYN flooding attacks and common mitigations," Internet Engineering Task Force, RFC 4987, 2007. (visited on 12/13/2016).

[39] R. Stewart, "Stream Control Transmission Protocol," Internet Engineering Task Force, RFC 4960, Sep. 2007. (visited on 11/16/2017).

[40] J. Lemon, "Resisting {SYN} flood {DoS} attacks with a {SYN} cache," in *BSDCon 2002*, 2002, pp. 89–98.

[41] C. Smith and A. Matrawy, "Comparison of operating system implementations of SYN flood defenses (Cookies)," in *2008 24th Biennial Symposium on Communications*, Kingston, ON, Canada: IEEE, Jun. 2008, pp. 243–246. DOI: 10.1109/BSC.2008.4563248.

[42] Cloudflare, *Understanding the Cloudflare Browser Integrity Check*, https://support.cloudflare.com/hc/en-us/articles/200170086-Understanding-the-Cloudflare-Browser-Integrity-Check, 2022. (visited on 04/29/2022).

[43] CAPTCHA, *The Official CAPTCHA Site*, http://www.captcha.net/, 2022. (visited on 04/29/2022).

[44] Google, *reCAPTCHA*, https://www.google.com/recaptcha/about/, 2022. (visited on 04/29/2022).

[45] T. Meunier, *Humanity wastes about 500 years per day on CAPTCHAs. It's time to end this madness*, http://blog.cloudflare.com/introducing-cryptographic-attestation-of-personhood/, May 2021. (visited on 04/29/2022).

[46] J. Beal and T. Shepard, "Deamplification of DoS Attacks via Puzzles," ResearchGate, 2004. (visited on 07/30/2023).

[47] S. Stidham, "Optimal control of admission to a queueing system," *IEEE Transactions on Automatic Control*, vol. 30, no. 8, pp. 705–713, Aug. 1985, ISSN: 1558-2523. DOI: 10.1109/TAC.1985.1104054.

[48] P. Naor, "The Regulation of Queue Size by Levying Tolls," *Econometrica*, vol. 37, no. 1, pp. 15–24, 1969, ISSN: 0012-9682. DOI: 10.2307/1909200. JSTOR: 1909200. (visited on 03/20/2023).

[49] J. Mirkovic, A. Hussain, S. Fahmy, P. Reiher, and R. K. Thomas, "Accurately Measuring Denial of Service in Simulation and Testbed Experiments," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 81–95, Apr. 2009, ISSN: 1545-5971. DOI: 10.1109/TDSC.2008.73.

[50] J. Mirkovic, A. Hussain, B. Wilson, S. Fahmy, P. Reiher, R. Thomas, W.-M. Yao, and S. Schwab, "Towards user-centric metrics for denial-of-service measurement," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, ser. ExpCS '07, New York, NY, USA: Association for Computing Machinery, Jun. 2007, 8–es, ISBN: 978-1-59593-751-3. DOI: 10.1145/1281700.1281708. (visited on 07/08/2022).

[51] J. Mirkovic, P. Reiher, S. Fahmy, R. Thomas, A. Hussain, S. Schwab, and C. Ko, "Measuring Denial Of Service," in *Proceedings of the 2Nd ACM Workshop on Quality of Protection*, ser. QoP '06, New York, NY, USA: ACM, 2006, pp. 53–58, ISBN: 978-1-59593-553-3. DOI: 10.1145/1179494.1179506. (visited on 01/18/2018).

[52] J. J. Echevarria, P. Garaizar, and J. Legarda, "An experimental study on the applicability of SYN cookies to networked constrained devices," *Software: Practice and Experience*, vol. 48, no. 3, pp. 740–749, 2018, ISSN: 1097-024X. DOI: 10.1002/spe.2510. (visited on 10/01/2021).

[53] D. Scholz, S. Gallenmüller, H. Stubbe, B. Jaber, M. Rouhi, and G. Carle, *Me Love (SYN-)Cookies: SYN Flood Mitigation in Programmable Data Planes*, Mar. 2020. DOI: 10.48550/arXiv.2003.03221. arXiv: 2003.03221 [cs]. (visited on 07/08/2022).

[54] M. A. Noureddine, A. M. Fawaz, A. Hsu, C. Guldner, S. Vijay, T. Başar, and W. H. Sanders, "Revisiting Client Puzzles for State Exhaustion Attacks Resilience," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2019, pp. 617–629. DOI: 10.1109/DSN.2019.00067.

[55] S. DeLaughter and K. Sollins, "Context Matters: Accurately Measuring the Efficacy of Denial-of-Service Mitigations," in *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*, ser. CSET '22, New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 91–99, ISBN: 978-1-4503-9684-4. DOI: 10.1145/3546096.3546109. (visited on 03/20/2023).

[56] M. Naor and M. Yung, "Universal one-way hash functions and their cryptographic applications," in *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, ser. STOC '89, New York, NY, USA: Association for Computing Machinery, Feb. 1989, pp. 33–43, ISBN: 978-0-89791-307-2. DOI: 10.1145/73007.73011. (visited on 07/19/2023).

[57] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, "Cryptographic Hash Functions: A Survey," Survey, Center for Computer Security Research, Department of Computer Science, University of Wollongong, 1995. (visited on 07/27/2023).

[58] R. Sobti and G. Geetha, "Cryptographic Hash Functions: A Review," *International Journal of Computer Science Issues*, vol. 9, no. 2, Mar. 2012, ISSN: 1694-0814. (visited on 07/27/2023).

[59] C. Dwork and M. Naor, "Pricing via Processing or Combatting Junk Mail," in *Advances in Cryptology — CRYPTO' 92*, E. F. Brickell, Ed., vol. 740, Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 139–147, ISBN: 978-3-540-57340-1. DOI: 10.1007/3-540-48071-4_10. (visited on 09/16/2020).

[60] B. Laurie and R. Clayton, ""Proof-of-Work" Proves Not to Work," *3rd Annual Workshop on Economics and Information Security*, p. 9, 2004.

[61] S. Nakamoto, "Bitcoin: A peer-to-peer electronic Cash System," bitcoin.org, 2008. (visited on 07/27/2023).

[62] J. Li, N. Li, J. Peng, H. Cui, and Z. Wu, "Energy consumption of cryptocurrency mining: A study of electricity consumption in mining cryptocurrencies," *Energy*, vol. 168, pp. 160–168, 2019.

[63] J. Sedlmeir, H. U. Buhl, G. Fridgen, and R. Keller, "The Energy Consumption of Blockchain Technology: Beyond myth," *Business & Information Systems Engineering*, vol. 62, no. 6, pp. 599–608, 2020. (visited on 07/19/2023).

[64] D. Clark, "The Design Philosophy of the DARPA Internet Protocols," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '88, New York, NY, USA: ACM, 1988, pp. 106–114, ISBN: 978-0-89791-279-2. DOI: 10.1145/52324.52336. (visited on 12/13/2016).

[65] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, Nov. 1984, ISSN: 0734-2071. DOI: 10.1145/357401.357402. (visited on 01/31/2019).

[66] *eBPF*, https://ebpf.io. (visited on 03/27/2023).

[67] L. Torvalds, *Linux kernel source tree*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c04c0d2b968ac45d
(visited on 07/14/2023).

[68] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed packet switching for local computer networks," *Communications of the ACM*, vol. 19, no. 7, pp. 395–404, Jul. 1976, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/360248.360253. (visited on 06/26/2021).

[69] W. Eddy, "Transmission Control Protocol (TCP)," Internet Engineering Task Force, Request for Comments RFC 9293, Aug. 2022. DOI: 10.17487/RFC9293. (visited on 07/18/2023).

[70] F. Gont and S. Bellovin, "Defending against Sequence Number Attacks," Internet Engineering Task Force, Request for Comments RFC 6528, Feb. 2012. DOI: 10.17487/RFC6528. (visited on 04/26/2022).

[71] J. Postel, "Transmission Control Protocol," Internet Engineering Task Force, RFC 793, 1981.

[72] K. A. Hua, N. Jiang, J. Kuhns, V. Sundaram, and C. Zou, "Redundancy control through traffic deduplication," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, Apr. 2015, pp. 10–18. DOI: 10.1109/INFOCOM.2015.7218362.

[73] K. Akpınar and K. A. Hua, "Deduplication overlay network," in *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, Oct. 2017, pp. 1–9. DOI: 10.1109/NCA.2017.8171369.

[74] M. Ruiz, G. Sutter, S. López-Buedo, J. F. Zazo, and J. E. López de Vergara, "An FPGA-based approach for packet deduplication in 100 gigabit-per-second networks," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec. 2017, pp. 1–6. DOI: 10.1109/RECONFIG.2017.8279776.

[75] M. Yoon, "A constant-time chunking algorithm for packet-level deduplication," *ICT Express*, vol. 5, no. 2, pp. 131–135, Jun. 2019, ISSN: 2405-9595. DOI: 10.1016/j.icte.2018.05.005. (visited on 07/30/2023).

[76] J. Martin, J. Burbank, W. Kasch, and P. D. L. Mills, "Network Time Protocol Version 4: Protocol and Algorithms Specification," Internet Engineering Task Force, Request for Comments RFC 5905, Jun. 2010. DOI: 10.17487/RFC5905. (visited on 07/30/2023).

[77] P. Hsieh, *Hash Functions*, http://www.azillionmonkeys.com/qed/hash.html, Blog, 2004. (visited on 06/13/2023).

[78] M. Kucherawy, D. Crocker, and T. Hansen, "DomainKeys Identified Mail (DKIM) Signatures," Internet Engineering Task Force, Request for Comments RFC 6376, Sep. 2011. DOI: 10.17487/RFC6376. (visited on 06/15/2023).

[79] E. Blanton, V. Paxson, and M. Allman, "TCP Congestion Control," Internet Engineering Task Force, Request for Comments RFC 5681, Sep. 2009. DOI: 10.17487/RFC5681. (visited on 07/14/2023).

[80] nsnam, *Ns-3 Network Simulator*, https://www.nsnam.org/. (visited on 07/20/2023).

[81] Mininet Project Contributors, *Mininet*, http://mininet.org/. (visited on 07/20/2023).

[82] USC Information Sciences Institute and University of Utah, *DeterLab: Cyber-Defense Technology Experimental Research Laboratory*, https://www.isi.deterlab.net/index.php. (visited on 05/31/2020).

[83] Red Hat, *Ansible*, https://www.ansible.com. (visited on 07/27/2023).

[84] NSF, *NSF Future Internet Architecture Project*, http://www.nets-fia.net/. (visited on 12/14/2016).

[85] *Named Data Networking*, https://named-data.net/. (visited on 12/14/2016).

[86] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, k. claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named Data Networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, p. 8, 2014.

[87] *MobilityFirst FIA Overview*, http://mobilityfirst.winlab.rutgers.edu/. (visited on 12/14/2016).

[88] A. Farhat, S. DeLaughter, and K. Sollins, "Measuring and Analyzing DoS Flooding Experiments," in *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*, ser. CSET '22, New York, NY, USA: Association for Computing Machinery, Aug. 2022, pp. 81–90, ISBN: 978-1-4503-9684-4. DOI: 10.1145/3546096.3546105. (visited on 08/28/2023).

[89] S. E. Deering and B. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," Internet Engineering Task Force, Request for Comments RFC 1883, Dec. 1995. DOI: 10.17487/RFC1883. (visited on 07/18/2023).

[90] Google, *Google IPv6 Statistics*, https://www.google.com/intl/en/ipv6/statistics.html#tab=ipv6-adoption. (visited on 07/10/2023).