SurgeProtector: Mitigating Temporal Algorithmic Complexity Attacks using Adversarial Scheduling

Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, Justine Sherry
Carnegie Mellon University

Abstract

Denial-of-Service (DoS) attacks are the bane of public-facing network deployments. Algorithmic complexity attacks (ACAs) are a class of DoS attacks where an attacker uses a *small* amount of adversarial traffic to induce a *large* amount of work in the target system, pushing the system into overload and causing it to drop packets from innocent users. ACAs are particularly dangerous because, unlike volumetric DoS attacks, ACAs don't require a significant network bandwidth investment from the attacker. Today, network functions (NFs) on the Internet must be designed and engineered on a case-by-case basis to mitigate the debilitating impact of ACAs. Further, the resulting designs tend to be overly conservative in their attack mitigation strategy, limiting the innocent traffic that the NF can serve under common-case operation.

In this work, we propose a *more general* framework to make NFs resilient to ACAs. Our framework, SurgeProtector, uses the NF's scheduler to mitigate the impact of ACAs using a very traditional scheduling algorithm: Weighted Shortest Job First (WSJF). To evaluate SurgeProtector, we propose a new metric of vulnerability called the Displacement Factor (DF), which quantifies the 'harm per unit effort' that an adversary can inflict on the system. We provide novel, adversarial analysis of WSJF and show that *any* system using this policy has a worst-case DF of only a small constant, where traditional schedulers place no upper bound on the DF. Illustrating that SurgeProtector is not only theoretically, but practically robust, we integrate SurgeProtector into an open source intrusion detection system (IDS). Under simulated attack, the SurgeProtector-augmented IDS suffers 90-99% lower innocent traffic loss than the original system.

CCS Concepts

• **Networks** → **Denial-of-service attacks**; **Packet scheduling**.

Keywords

Network Security, Algorithmic Complexity Attacks, Adversarial Scheduling, SurgeProtector, Pigasus, WSJF

ACM Reference Format:
Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, Justine Sherry. 2022. SurgeProtector: Mitigating Temporal Algorithmic Complexity Attacks using Adversarial Scheduling. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3544216.3544250

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9420-8/22/08.
https://doi.org/10.1145/3544216.3544250

1 Introduction

Network functions are vulnerable targets for *algorithmic complexity attacks* (ACAs) [12]. With an ACA, an attacker crafts a carefully-designed input that requires a *small amount of network and compute resources for the attacker to produce*, and yet consumes a *large amount of compute resources at the target system*. Given a sufficient request rate, an attacker can drive the victim into overload, causing it to drop requests from the innocent, intended users of the service. ACAs are especially dangerous when compared to traditional 'volumetric' Denial-of-Service (DoS) attacks. In a volumetric attack, an attacker must invest the necessary resources to, *e.g.*, produce 100M packets/sec in order to overload an intrusion detection system (IDS) which is provisioned to serve 100M packets/sec; conversely, with an ACA, an attacker with only modest resources can overload a much more powerful service (say, producing only 1Mpps to overwhelm the same 100Mpps-provisioned IDS).

In this paper, we evaluate ACAs via a novel measure of vulnerability called the *Displacement Factor* (DF). The key idea behind the DF is to measure the ratio of innocent traffic displaced by an attacker ('harm') to the attacker's own bandwidth investment ('effort'). A DF of 0 implies that no innocent traffic is ever displaced, and a DF of 100 implies that for every 1 bps of attack traffic, 100 bps of innocent traffic are displaced. A 2012 published attack on IDS regular expression engines achieved a DF of 8 [1], and a 2019 published attack on Open vSwitch exploiting the Tuple-Space-Search (TSS) algorithm [13] achieved DFs as high as 12,000!

As we will discuss in §2, ACAs are particularly challenging to mitigate in NFs. In order to be resilient against ACAs, state-of-the-art solutions (a) must be designed on a case-by-case basis, and (b) limit the traffic that the NF can serve under normal operation. For example, it is common practice for regular-expression based DPI engines to limit how many states in the regular expression DFA a particular packet or flow may traverse [48]. This prevents an attacker from wasting compute cycles, thereby reducing the DF. However, this also prevents the network operator from deploying particularly complex rules, limiting the NF's ability to serve legitimate traffic which traverses a large number of DFA states, even under normal operation (i.e., when the NF is operating below maximum capacity and is easily able to service such traffic).

In this paper we ask: is there a *general* approach for mitigating algorithmic complexity attacks on NFs which does *not* limit the types of rules and traffic that can be serviced under normal operation?

We are inspired by general solutions to ACAs in the traditional systems literature as we aim towards a general – rather than NF-by-NF – solution. In cluster-compute frameworks [27, 34, 51] and operating systems [8, 32], ACAs are less of a concern because performance isolation techniques prevent the resource usage of one user from impacting that of another. In these systems, the *scheduler* divides

compute time evenly between users, and even if a user submits an expensive job for servicing, other users still receive their 'fair share' of service time. Unfortunately, as we discuss in §4.2, in the networking setting, a Fair Queueing (FQ) [17] scheduler with this same approach can be easily exploited by an attacker who generates traffic which appears as if it is coming from multiple users, fooling the scheduler into allocating more service time to the attacker.

We find that a 'familiar friend' from the scheduling literature is, surprisingly, an effective mitigation strategy against ACAs. Weighted Shortest Job First (WSJF) [11] naturally discards costly packets when the system is overloaded, yet under normal operation, it will eventually serve all packets, even those with lengthy service times.

While WSJF is an old algorithm, our analysis of WSJF in the context of ACAs is novel. *In §4.4, we prove that WSJF enforces a DF with an upper bound of 1, regardless of the DF of the underlying algorithms, the load on the server, and the parameters of the innocent packet and job size distributions.* In other words, WSJF ensures that, in order to displace 1 bps of innocent traffic, the adversary must inject *at least* 1 bps of their own bandwidth into the attack, significantly mitigating the impact of ACAs. In comparison, traditional First Come First Served (FCFS) and Fair Queueing (FQ) schedulers do not place any upper bound on the DF.

We bring our theoretical results into practice by building SurgeProtector, an implementation of WSJF for NFs that we integrate into an open-source intrusion detection system (IDS). Doing so required addressing several pragmatic challenges. First, WSJF assumes that per-packet processing times are known *a priori*, which may not be practical in the context of real data-structures and algorithms. Second, the theory behind SurgeProtector assumes packets can be arbitrarily reordered, but we know that TCP performs poorly in the face of reordering. Finally, SurgeProtector requires a priority queue to schedule in WSJF order – exposing yet another attack surface. We describe our implementation of the SurgeProtector scheduler in the context of the Pigasus IDS [58] and discuss how it addresses all of these challenges in §5.

Then, in §6, we evaluate SurgeProtector both in simulation and our empirical testbed.[1] Although SurgeProtector upper-bounds the DF to 1, in practice, we see a worst-case DF of at most 0.4 – that is, to displace 1 bps of innocent traffic, the attacker must invest at least 2.5 bps of their own bandwidth into the attack – where previously the DF had been over 100. Hence, compared to the baseline IDS implementation, the SurgeProtector-augmented IDS yields 90-99% lower loss of innocent traffic under a worst-case attack.

The prospect of using adversarial scheduling to mitigate ACAs opens up several interesting theoretical and practical questions, and we are only able to answer some of them. Perhaps the most important open questions pertain to how to predict job sizes *a priori*; SurgeProtector ultimately relies on heuristics for this task, but we believe that a thorough analysis of efficient, adversary-proof heuristics remains ripe for exploration. Thus, in §7, we describe current limitations and various open questions (regarding heuristics, fairness, *etc.*). Finally, we describe related work in §8, and conclude in §9.

2 Background and Motivation

Algorithmic complexity attacks target a system's underlying algorithms and/or data-structures, using specially-crafted inputs to trigger the system's worst-case behavior [1, 4, 12, 47]. While the attacker's input pattern(s) and the resulting behavior may vary from design to design, the ultimate goal of these attacks is the same: to overload the system with large amounts of wasteful work, inhibiting its ability to serve innocent user traffic.[2] The key difference between an ACA and a traditional volumetric DoS attack is that in an ACA, the attacker can induce the system to perform a *large* amount of wasteful work by introducing a *small* input that costs little to produce. In a volumetric DoS attack, the attacker must craft a *large* amount of input to overload the system, which requires the investment of physical resources to produce this traffic. Colloquially, an ACA provides 'more bang for one's buck.'

Example: Consider the following, simplified example drawn from Pigasus [58]. Pigasus is a hybrid FPGA+CPU, 100Gbps IDS, and it implements partial TCP reassembly in order to detect attacks that span across multiple packets in a TCP bytestream. As shown in the Figure 1, Pigasus stores packets from out-of-order flows in a linked list. When a packet corresponding to an out-of-order flow arrives, the reassembly engine traverses its linked list to find the appropriate insertion location (using the packet sequence number), performs insertion, and, if possible, releases any in-order segments.

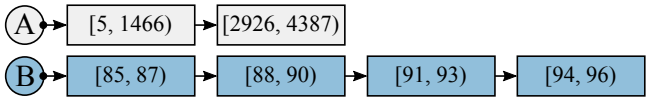


Figure 1: TCP reassembly using a linked list [58]. Each node in the list represents a range of packet sequence numbers.

Let us assume for the sake of exposition that most connections look like flow A in Figure 1, with exactly two packets in the linked list and only one 'gap' in the sequence number space. When a re-transmitted or re-ordered packet arrives to fill in a gap in the sequence number space (*e.g.*, a packet with sequence number 1466 in flow A), it takes two iterations of pointer-chasing to reach the right index in the linked list.

To mount an ACA, an attacker might transmit a sequence of packets leading to a scenario more like flow B: should a packet arrive with index 93, it would take four iterations of pointer chasing – or twice as many cycles as in the typical case – to fill in the sequence number gap.

We refer to the amount of work the system performs to process a packet as the packet's *job size*, with the average 'innocent' packet's job size J_I and attack job sizes averaging J_A. Now let us assume the system is operating at capacity: there are some C packets per second arriving at the system, with an average of J_I job size per packet. If some of those packets are instead sized $J_A > J_I$, the system will be unable to keep up with the offered load and be forced to drop some packets. If an attacker injects one packet of sized $J_A = 10$, and all other packets are $J_I = 2$, then the system will be forced to drop 5 innocent packets in order to process the additional attack packet. In our simulations with Pigasus (§6), we found that in practice, an attacker could force Pigasus' reassembly engine to drop roughly 300 innocent bits for every bit of input attack traffic.

[1] Artifacts are available at https://github.com/cmu-snap/SurgeProtector

[2] In this paper, we focus on *temporal* ACAs, in which an attacker crafts system inputs which are computationally expensive to process, consuming compute cycles that could be used for innocent inputs. There are some attacks where adversarial inputs *e.g.*, aim to poison datastructure contents [23], but are not themselves computationally expensive to process. These attacks are sometimes also referred to ACAs, but they are not the focus of this work.

Unfortunately, the literature is full of examples of NFs vulnerable to ACAs: For example, in 2020, researchers showed that they could slow the popular open-source software switch, Open vSwitch, to support only 1% of its typical throughput by offering a small 1 Mbps attack stream designed to exploit algorithmic complexity [14]. The attack exploited a well-known vulnerability in the Tuple-Space Search (TSS) [49] algorithm for packet classification known as 'Tuple Space Explosion' (TSE) [13, 15].

In 2018, [50] identified a vulnerability in the Linux kernel's TCP reassembly logic. Although the Linux implementation uses a more sophisticated data-structure to manage out-of-order flows (Red-Black Trees), the bug allowed malicious peers to consume an excessive number of CPU cycles using specially-crafted inputs. The bug was addressed by a patch that streamlined processing enough to render the attack 'not critical' [19]; while this may be sufficient for the current line-rate supported by kernel networking, the vulnerability will inevitably resurface alongside the next generation of line-rates.

An entire sub-literature of research [1, 4, 12, 47] addresses attacks on deep-packet inspection (DPI) engines (*e.g.*, Pigasus [58], Snort [38], Suricata [16]) via Regular expression Denial of Service (ReDoS). A ReDoS attack crafts packets with payloads that are carefully designed to traverse multiple states in regular expression automata – the more states the packet triggers in the automata, the larger the J_A for that packet. Previous work has shown that an attacker responsible for only 10% of the traffic entering a regular expression engine can slow down legitimate traffic by up to 500% [1]. The literature is rife with other examples: ACAs that exploit decompression algorithms, sorting, hash tables, *etc.* [26, 29, 35, 36].

We note that some attacks are referred to as ACAs which are not *temporal*, but rather *spatial* in nature. For instance, an attacker might exploit a key-value store that uses separate chaining to resolve hash collisions [23] by injecting a large number of their own key-value pairs into the store. This increases the load factor of the underlying hash table, driving up the job size for *all* traffic – not just the attacker's – arriving afterwards. In this work, we focus exclusively on temporal ACAs (i.e., assume a threat model where the attacker can control the job sizes of their own packets, but cannot influence the job size distribution for innocent traffic).

Resource isolation is insufficient to prevent ACAs in a networked setting: Many systems aim to shield users from the actions of other (potentially malicious) users by allocating each one a fixed slice of the shared resource (i.e., *resource isolation*). Unfortunately, the networking equivalent to resource isolation – *fair queueing* [17] – is trivially circumvented and hence middleboxes and NFs are especially vulnerable. A fair queueing device schedules packets for processing in such a way as to divide service time equally between classes of traffic – service time might be divided evenly by network connection, by class of traffic (*e.g.*, HTTP vs VOIP traffic), or by sender. At first glance, it might appear that this would prevent an attacker from consuming more than their 'fair share' of processor time. But, unfortunately, on the Internet, attackers have numerous ways to easily *spoof* the source IP address of their traffic – leading to the appearance that the attack traffic originates from *multiple* users.

Existing, application-specific solutions lead to undesirable tradeoffs: Most mitigation techniques for ACAs in NFs instead turn to shrinking the gap between J_I, the innocent job size, and J_A, the worst-case attack job size. While this approach is state-of-the-art, it leads to undesirable trade-offs between common-case usability in exchange for ACA resilience.

Returning to the flow reassembly case, one might enforce that no linked list ever extends further than a chain of four packets, and if additional out-of-order packets arrive, the flow is simply reset. This approach mitigates the ACA: where a malicious packet might have led to the loss of n innocent packets in the base design, we can bound J_A to bring it closer to J_I and limit the malicious packet to only cause a loss of $m < n$ packets.

Unfortunately, imposing a maximum length on the reassembler limits usability in the common case: we reduce J_A, but we also limit the NF's ability to handle innocent highly out-of-order flows, *even in scenarios where the NF has excess capacity and can feasibly service them.* Thus, the NF designer is left with two equally unappealing alternatives. They can either set a higher limit on J_A, allowing the NF to service a wider range of flows but leaving it more vulnerable to ACAs, or they can set a lower limit on J_A, thereby sacrificing the ability to serve certain innocent flows for the sake of higher ACA resilience.

As we will discuss in §8, NFs today come with a variety of such patches in an effort to restrict J_A, and sacrifice some property or the other (*e.g.*, common-case performance or memory efficiency) in exchange for ACA resilience. Additionally, the application-specific nature of these patches means that there is no *general* solution for mitigating ACAs – every patch must be constructed from scratch for each new ACA. This motivates our search for an attack mitigation strategy that is both general and obviates the need to make undesirable tradeoffs in order to achieve resiliency against ACAs.

3 Problem Definition

In order to facilitate a first-principles analysis of algorithmic complexity attacks, we start by formulating a theoretical model to capture the dynamics of packets and jobs in §3.1. Next, we characterize the adversary's capabilities and our threat model in §3.2. In §3.3, we formally define the *Displacement Factor* (DF). In §4 we use these foundations to demonstrate how scheduling can be used to mitigate ACAs.

3.1 System Model

Packets and jobs: At the heart of our abstraction is an NF that serves packets appearing on an ingress link of capacity R Gbps. Each packet requires a certain amount of time to be processed (*e.g.*, due to computation, I/O, memory lookups, *etc.*), and thus can be characterized by two independent variables: a packet size (in bits) and a job size (in seconds). For convenience, we also tag each packet with a class: class I packets correspond to innocent traffic and class A packets correspond to adversarial traffic; however, note that this tag is only relevant for the purpose of our analysis, and is not visible to the underlying system.

We assume that packets belonging to innocent traffic follow certain packet and job size distributions, with P and J denoting continuous random variables sampled from these distributions, respectively. Let $f_P(p)$ and $f_J(j)$ denote their probability density functions (pdf),[3] and $\mathbb{E}[P]$ and $\mathbb{E}[J]$ denote the corresponding expectations. Table 1 contains a summary of the notations used in the model.

[3] In general, the packet size and job size may be correlated, and we use $f_{P,J}(p, j)$ to denote the joint pdf.

Notation	Description
R	Link capacity (in Gbps)
P	Packet size of class I traffic (random variable)
J	Job size of class I traffic (random variable)
$f_P(p)$	Probability density function of packet size P
$f_J(j)$	Probability density function of job size J
$P_{\min}, P_{\max}$	Minimum, maximum packet sizes
$J_{\max}$	Maximum job size
r_I	Input traffic rate (in Gbps) for class I traffic
$r_{\max}$	Maximum serviceable traffic rate
o_I	Output traffic rate (in Gbps) for class I traffic
$\alpha(r_I)$	Displacement Factor (DF)

Table 1: Summary of notations used in the model.

Goodput: Let r_I denote the input traffic rate (in Gbps) for class I traffic on the ingress link. For simplicity, we assume that packet arrivals have a constant inter-arrival time; i.e., the inter-arrival time is $\frac{\mathbb{E}[P]}{r_I}$ seconds for innocent traffic. We define the system *goodput*, denoted as o_I, as the output traffic rate corresponding to class I traffic; i.e., the *useful* throughput that the system can sustain. Note that the system is designed to serve innocent traffic, and the maximum serviceable traffic rate without dropping packets is given by $r_{max} = \frac{\mathbb{E}[P]}{\mathbb{E}[J]}$ (in Gbps). Thus, in the absence of any adversarial traffic, the goodput $o_I = r_I$ when $r_I \le r_{\max}$. The system model is depicted in Figure 2.

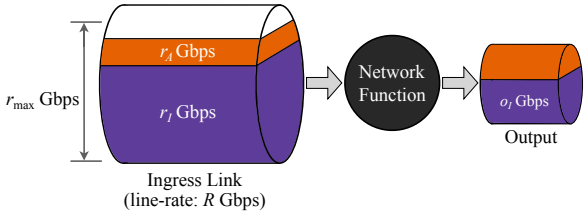


Figure 2: System model.

3.2 Threat Model

In order to model algorithmic complexity attacks, we allow a rate-limited adversary to inject a stream of *adversarial* (class A) traffic into the ingress link. Let r_A denote the input traffic rate for class A traffic. To enforce line-rate semantics, we impose the constraint $r_I + r_A \le R$. Our threat model assumes an attacker that is overpowered relative to what we believe a practical attacker is capable of. In particular, we assume that the adversary is aware of all aspects of the underlying system ('transparent' model), as well as the innocent packet and job size distributions, and always uses the *optimal attack strategy*. In particular, the adversary crafts packets with the best choice of packet size and job size to maximize the harm to the system, where the harm is measured by reduction in goodput as defined in §3.3. The adversary is *not* capable of: (a) inspecting individual packets as they appear on the ingress link, (b) affecting the job sizes of class I packets (*e.g.*, by tainting shared state), or (c) amplifying their attack bandwidth using other means (*e.g.*, reflection-based amplification).

3.3 Quantifying Vulnerability

We first measure the harm induced by the adversary using the volume of innocent traffic 'displaced' under a given attack traffic input rate r_A. Specifically, we write the goodput o_I as $o_I(r_I, r_A)$ here to explicitly express its dependence on r_I and r_A. Then the volume of innocent traffic displaced is $o_I(r_I, 0) - o_I(r_I, r_A)$, i.e., how far the goodput deviates from the goodput in the absence of an adversary ($r_A = 0$).

We then quantify the vulnerability of the system using the *Displacement Factor* (DF), α, defined as the adversary's payoff relative to the amount of resources they invest:

$$DF = \frac{\text{Innocent traffic displaced (Gbps)}}{\text{Attack bandwidth used (Gbps)}}$$

A DF of 5 means an attacker can force the NF to drop 5 bits of innocent traffic for every 1 bit of attack traffic provided. More formally, we can write the DF as follows:

$$\alpha(r_I) = \sup_{r_A} \frac{o_I(r_I, 0) - o_I(r_I, r_A)}{r_A}. \quad (1)$$

Here we take the supremum over the attack traffic rate r_A to capture the adversary's most efficient attack.

4 Mitigating ACAs using Scheduling

In this section, we demonstrate how scheduling can be used to effectively mitigate ACAs in a networked setting. As a starting point, we first consider two commonly-used scheduling policies, First-Come First-Served (FCFS) and Fair Queueing (FQ). In §4.1 and §4.2, we show that under both FCFS and FQ, the DFs become unbounded in some regimes of system parameters. Consequently, systems that use FCFS or FQ scheduling *cannot* rely on the scheduler to protect against ACAs.

To build intuition as to how a job-size based scheduling policy can limit the harm induced by the adversary, we then present a scheduling policy called Shortest Job First (SJF) in §4.3. We show that SJF has a DF upper bounded by a constant that is independent of $J_{\max}$, improving upon both FCFS and FQ; however, this constant grows as the average packet size for innocent traffic, $\mathbb{E}[P]$, increases. We then present Packet-Size Weighted Shortest Job First (WSJF) in §4.4, showing that WSJF further removes the dependence on $\mathbb{E}[P]$ and achieves a maximum DF of 1. Finally, we summarize SurgeProtector's theoretical guarantees in §4.5.

Due to space constraints, we merely provide the intuition behind each claim here, and defer all proofs to Appendix A.

4.1 First-Come First-Serve (FCFS)

As the name suggests, *First-Come First-Serve* (FCFS) serves jobs in the order that they appear on the ingress link. Under FCFS, in order to maximize harm, the adversary crafts packets with the smallest possible packet size, $P_{\min}$, and the largest possible job size, $J_{\max}$.

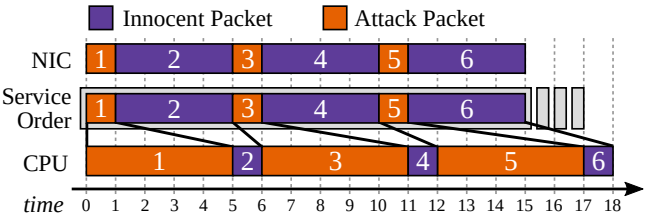


Figure 3: FCFS fails to protect against ACAs.

As depicted in Figure 3, using small-sized packets encoding large jobs enables an attacker to consume a significant fraction of CPU (i.e., service time) despite using only a small amount of NIC time (i.e.,

attack bandwidth), throttling goodput. Intuitively, this happens because FCFS serves jobs in the order of arrival regardless of their sizes. Therefore, if an adversary can craft packets with arbitrarily large job sizes, they can also reduce the traffic rate for innocent packets to an arbitrarily large degree. We show in Claim 1 below that the adversary can achieve unbounded DF under FCFS as $\frac{J_{\max}}{P_{\min}}$ becomes large.

Claim 1 (**DF of FCFS**). *Under FCFS, for any innocent input traffic rate r_I and any packet size and job size distributions, the Displacement Factor $\alpha_{FCFS}(r_I) \to +\infty$ as $\frac{J_{\max}}{P_{\min}} \to +\infty$.*

The detailed proof can be found in §A.1.

4.2 Fair Queueing

Fair Queueing (FQ) is a scheduling algorithm that is widely employed in switches and network processors. FQ and its variants (*e.g.*, WFQ, DRFQ) ensure that one or more shared resources (*e.g.*, network throughput, processor time, *etc.*) are evenly partitioned among a number of competing flows. While this scheme performs well when these flows are operated by good faith users seeking fair arbitration over a shared, limited resource, it does not translate well to the adversarial setting.

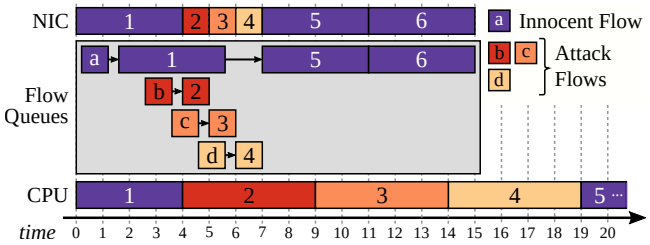


Figure 4: FQ fails to protect against ACAs. In steady-state, the attacker receives 75% of the total service time despite using a small attack bandwidth.

The fundamental problem is that FQ only guarantees equitability across *flows*, thereby allowing a malicious user to occupy a disproportionately high fraction of the shared resource(s) by spawning more flows. Further, using FQ at source IP granularity is also insufficient because of the possibility of source address spoofing. As depicted in Figure 4, using small-sized packets across a large number of flows enables an attacker to consume a significant fraction of service time using only a small amount of attack bandwidth. As we show in the proof for Claim 2, the DF under FQ ultimately scales with $\frac{J_{\max}}{P_{\min}}$, and, as in the case of FCFS, can become unbounded.

Claim 2 (**DF of FQ**). *Under FQ, for any innocent input traffic rate r_I and any packet size and job size distributions, the Displacement Factor $\alpha_{FQ}(r_I) \to +\infty$ as $\frac{J_{\max}}{P_{\min}} \to +\infty$.*

The detailed proof can be found in §A.2.

4.3 Shortest Job First (SJF)

FCFS's obliviousness to job sizes and FQ's focus on per-flow fairness leaves them both susceptible to ACAs. In order to prevent ACAs, we need a scheduling policy that considers job sizes without being vulnerable to flow inflation. *Shortest Job First* (SJF) is a popular policy for scheduling jobs in a non-preemptive system. As the name suggests, at any instant, SJF prioritizes the queued job with the smallest (initial) job size.

We show in Theorem 1 below that the DF under SJF is upper bounded by a small constant independent of both $J_{\max}$ and $f_J(j)$. The intuition behind why SJF works well is simple: if the adversary produces packets whose jobs are *too expensive* to process, they will simply be de-prioritized and never end up being served. Instead, if the adversary produces packets whose jobs are *too cheap*, they will fail to push the system into overload. As depicted in Figure 5, the attacker's optimal strategy is to pick a job size corresponding to a 'sweet spot' in the innocent job size distribution,[4] and use minimum-sized packets to inflate their packet rate (and, consequently, the total work injected into the system). This allows them to displace *some* innocent traffic, achieving a worst-case constant DF. As we show in the proof for Theorem 1, the DF under SJF scales as the ratio between the average packet size for innocent traffic, $\mathbb{E}[P]$, and the minimum packet size, P_{min}.

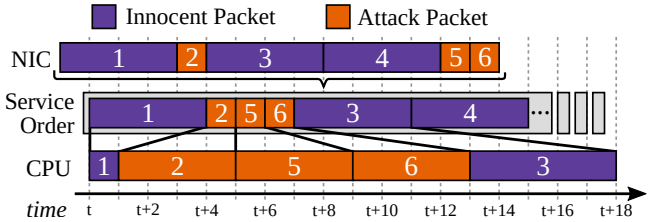


Figure 5: In order to exploit SJF, the attacker uses minimum-sized packets with a job size (i.e., CPU time) between that of packets 1 and 3. The attack packets (i.e., 2, 5, 6) are scheduled before more expensive ones (3, 4), pushing the system into overload and displacing packet 4.

Theorem 1 (**DF of SJF**). *Under SJF, for any innocent input traffic rate r_I and any packet size and job size distribution, the Displacement Factor is upper bounded as:*

$$\alpha_{SJF}(r_I) \leq \frac{\mathbb{E}[P]}{P_{\min}} \cdot \rho,$$

where $\rho = \min\left(\frac{r_I}{r_{max}}, 1\right) \in [0,\ 1]$ is the load on the system due to innocent traffic.

Unlike FCFS and FQ, SJF *does* impose an upper bound on the DF, limiting the extent that an attacker can cause harm to the system. We show in the detailed proof (§A.3) that SJF has an upper bound that depends on $\frac{\mathbb{E}[P]}{P_{min}}$, which is approximately a factor of 8 given typical innocent packet size distributions. We show that we can further improve this bound with *weighted* SJF in the next section.

4.4 Weighted Shortest Job First (WSJF)

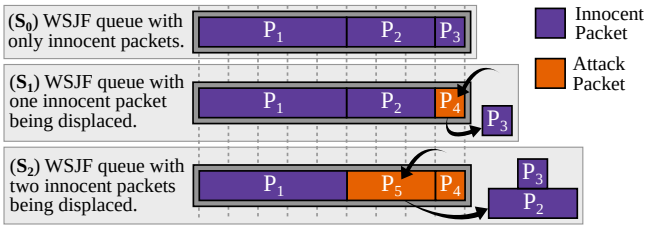
A fundamental limitation of the policies described so far is that they altogether ignore the *packet size* information encoded in an incoming packet. This enables an adversary to greatly inflate their job arrival rate using minimum-sized packets, leading to either an unbounded DF (in the case of FCFS and FQ), or one that scales inversely with P_{min} (in the case of SJF). Then, a natural question is: *can we do better by leveraging this readily-available information?*

Here, we propose to use *Packet-Size Weighted Shortest Job First* (WSJF), a variant of SJF that prioritizes the packet with the smallest *job-to-packet-size ratio*. We show in Theorem 2 below that the DF under WSJF is *at most 1*, which implies that for every 1 bps of innocent traffic that the adversary wishes to displace, they must consume

[4]For a formal characterization of the optimal attack strategy under SJF, please see Lemma 1 in Appendix A.

at least 1 bps of their own bandwidth. The intuition is that *WSJF minimizes the system's work-per-bit, thereby preventing an adversary from consuming a high fraction of processing cycles unless they invest a proportionally high bandwidth into the attack.*

To concretize this notion, consider the scenario depicted in the figure below. For the sake of simplicity, assume that all innocent packets have a job size of 1 unit time. Also assume that the system is operating at capacity, and that in steady-state, the WSJF queue contains 3 packets with packet sizes $P_1 = 5$, $P_2 = 3$, and $P_3 = 1$. Observe that WSJF would serve these packets in decreasing order of packet size (i.e., P_1 before P_2, and P_2 before P_3), corresponding to scenario S_0.



Consider an attacker that seeks to displace a single packet (i.e., with packet size P_3) from this queue with one of their own. In order to do this, the attacker *must* inject an attack packet of size $P_4 \geq P_3$ with a job size of 1 (scenario S_1); a smaller job size would introduce slack in the system load (allowing it to periodically serve innocent packets of size P_3 as well), while a smaller packet size would result in the attack packet never being served. Thus, the attacker is forced to inject as many bits as they wish to displace.

Suppose, instead, that the attacker wishes to displace *two* packets (with sizes P_2 and P_3). The attacker now has two options: they can either inject two packets with sizes $P_5 \geq P_2$ and $P_4 \geq P_3$ and unit job size each (scenario S_2), or a single packet of size $P_6 \geq (P_2 + P_3)$ and a job size of 2. Once again, the attacker's bandwidth investment matches or exceeds the displaced goodput. As we demonstrate in the proof for Theorem 2, this result generalizes to any load, as well as any job and packet size distributions of innocent traffic.

Theorem 2 (**DF of WSJF**). *Under WSJF, for any innocent input traffic rate r_I and any packet size and job size distribution, the Displacement Factor is upper bounded as:*

$$\alpha_{WSJF}(r_I) \leq \rho \leq 1,$$

where $\rho = \min\left(\frac{r_I}{r_{max}}, 1\right) \in [0,\ 1]$ is the load on the system due to innocent traffic.

The detailed proof can be found in §A.4.

4.5 SurgeProtector

SurgeProtector interposes a WSJF scheduler in front of variable-time modules within an NF (*e.g.*, the reassembler discussed in §2). WSJF meets both our initial goals of *generality* and *not limiting the innocent traffic that can be served.* First and foremost, it provides a provable upper bound on the DF that is *independent of the underlying algorithms.* WSJF is a drop-in solution that can be applied to any algorithm, and hence, it is general.[5] Second, where many ACA solutions, *e.g.*, drop packets from flows that are determined to be too expensive to process, WSJF guarantees that all connections will be served so long as there is system capacity to do so (i.e., it is starvation-free when the system is at or below capacity). Hence, WSJF does not place any limitations on innocent traffic under normal operation. In overload, the most computationally expensive packets *are* dropped,[6] but overall this *minimizes the rate of innocent traffic that is denied service.*

With WSJF as our chosen approach, we now turn to the challenges of integrating WSJF into a practical network function in the following section.

5 Implementation & Practical Issues

In order to validate our theoretical findings in the context of a real system, we incorporate SurgeProtector into the open-source Pigasus IDS [58]. A simplified block diagram of Pigasus is depicted in Figure 6. In §2, we briefly introduced the linked-list based design of Pigasus's FPGA-based TCP Reassembly engine (labelled ①), and demonstrated how it can be exploited by an adversary. It turns out that a second component of the IDS – the CPU-side Full Matcher (labelled ②) – is also vulnerable to a different type of complexity attack.

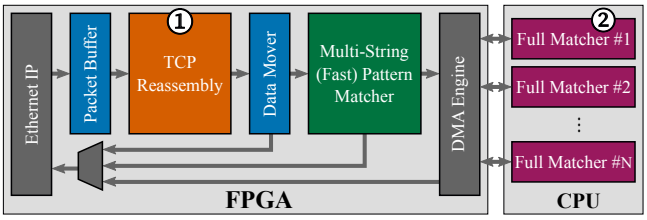


Figure 6: The simplified Pigasus IDS pipeline.

We begin in §5.1 with a brief overview of the two vulnerable components in Pigasus that we sought to protect. Over the course of implementing SurgeProtector, we encountered the following three important practical challenges. First, how do we predict job sizes? Second, since flow reordering is often undesirable, how do we guarantee in-order delivery for packets of the same flow? Finally, how do we ensure that the scheduler itself does not present a target for ACAs? We frame the implementation details of the SurgeProtector scheduler in the context of these three questions (§5.2 - §5.4).

5.1 Overview of Vulnerable Components

FPGA-based TCP Reassembly: Recall that the goal of TCP reassembly is to reconstruct an in-order TCP bytestream from a sequence of out-of-order packets. The Pigasus reassembler, which is FPGA-based, prioritizes memory efficiency, and employs a linked list-based design to manage out-of-order flow state. While this achieves excellent memory utilization, the worst-case linear complexity of linked-list operations makes it susceptible to ACAs.

An example of this is depicted in Figure 7. When a new packet arrives (with PSN range [35, 50) in the example below), the reassembler

[5] This assumes, for the moment, *a priori* knowledge of the packet processing time, which must be calculated based on the underlying algorithm. We return to address this point in more detail in §5.2.

[6] At this point, one might wonder: if WSJF drops the most expensive jobs in overload, why doesn't the adversary simply use less expensive jobs, thereby cajoling the scheduler into exclusively serving their traffic (*e.g.*, if innocent packets have a job size of 10 units, the adversary uses packets with a job size of 1 unit)? From an adversarial perspective, this turns out be an inefficient strategy; the adversary must now send 10X the number of packets to displace innocent traffic, corresponding to 10X as much attack bandwidth. In particular, this devolves the DoS attack into a *volumetric* one, which defeats the purpose of using an ACA in the first place.

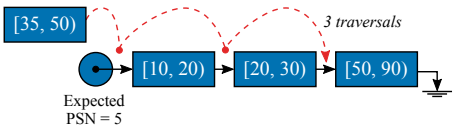


Figure 7: Linked-list state for an out-of-order flow.

linearly scans the list and inserts the node at the appropriate position. In order to exploit this, an attacker crafts *highly out-of-order flows*, linearly increasing the number of traversals required for each subsequent attack packet. Finally, they use *minimum-sized packets* (with a 1-byte TCP payload) to inflate their packet arrival rate, maximizing the work injected into the system.

CPU-based Full Matching: As a signature-based IDS, Pigasus identifies malicious flows by comparing packet payloads against a database of known attack signatures ('rules'). To achieve high performance, it does so in two stages: it first uses a number of fast filters in hardware (i.e., the Multi-String Pattern Matcher) to quickly filter out innocent traffic; it then relays the remaining (small) fraction of possibly-malicious traffic to the CPU to perform more expensive regex analysis ('Full Matching' [58]). While Pigasus' first stage operates in constant time (and hence is not vulnerable to ACAs), the CPU-side Full Matching stage involves variable-time computation that is also input-dependent, making it vulnerable to ACAs.

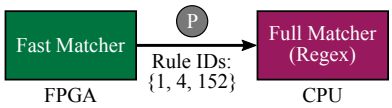


Figure 8: Pigasus Full Matching pipeline.

Pigasus's Full Matching pipeline is depicted in Figure 8. During the first stage, in addition to filtering out innocent packets, Pigasus also generates a list of candidate rules that the packet may ultimately match on. It then sends this list, along with the packet payload, to the CPU for processing. The CPU sequentially processes each rule in the list, stopping at the first rule that results in a match. The processing result (i.e., indicating whether to drop or forward the packet) is subsequently relayed back to the FPGA.

An attacker can exploit this by crafting attack packets that either: (a) result in a *large number of matches in the Fast Matching stage* (requiring the Full Matcher to evaluate many rules), (b) *trigger a regex search with super-linear runtime* in the Full Matcher (i.e., ReDoS-style attacks [9, 16, 56]), or both.

5.2 Predicting Job Sizes

SurgeProtector schedules packets based on job sizes, but, in practice, the time required to process a packet is not known *a priori*. A common approach to solve this problem – and one we employ in this work – is to use *heuristics* for job size estimation [31, 33]. In particular, we use the following heuristics to estimate job sizes for our target applications:

TCP Reassembly: a packet's job size is estimated as the *length of the out-of-order linked-list for the corresponding flow*. Despite its simplicity, this heuristic has two salient properties: first, since the number of traversals can never exceed the length of the linked-list, the estimate always *upper-bounds* a given packet's true job size; second, since the heuristic is computed on a per-flow basis, the adversary cannot affect the quality of estimates for innocent flows.

Full Matching: if K denotes the list of candidate rules identified by the fast matching stage, then the job size is estimated as $J = \Sigma_{k \in K}(z_k \cdot p)$, where z_k denotes the *maximum job-size-to-packet-size ratio observed for rule k thus far*, and p denotes the packet payload size. By using historical run-time data as feedback, the heuristic function 'learns' which rules are prone to complexity attacks and selectively deprioritizes them.

We implement and evaluate SurgeProtector using both these heuristics in §6.1. It is worthwhile to note that neither of these heuristics is 'ideal' in a theoretical sense. For example, in the case of TCP Reassembly, there *may* exist innocent TCP flows on the Internet for which the heuristic consistently overestimates job sizes by a significant margin, allowing the attacker to unfairly displace them. Similarly, in the case of Full Matching, an attacker *may* be able to manipulate the outcome of the heuristic for every rule, potentially causing large prediction errors for subsequent innocent packets.

In practice, this does not appear to be the case. For instance, in the case of TCP Reassembly, the heuristic yields accurate job size estimates for the vast majority of TCP flows, limiting the additional harm that an adversary can induce. Similarly, in the case of Full Matching, most rules don't have large variance in their job-size-to-packet-size ratios. We explore this further in §6.2, where we empirically evaluate the effect of using heuristics on SurgeProtector's DF upper-bound. *Empirically, we find that for both applications, the adversary's DF increases by no more than 5% of the upper-bound even when the adversary has perfect knowledge of the actual and heuristic-estimated job size distributions.* We leave the exploration of adversary-proof job size heuristics for arbitrary NFs to future work (§7).

5.3 Keeping (TCP) Flows In-Order

Keeping packets within the same TCP flow in order is necessary to avoid degrading application performance [5, 18, 30, 45, 58]. While FCFS and FQ (along with its variants) guarantee that same-flow packets are served in-order, SJF and WSJF do not. In this section, we explore how to augment SurgeProtector to provide in-order service.

As a natural starting point, consider the following extension to WSJF, which we will refer to as *WSJF Head-of-Queue* (WSJF-HoQ). This policy maintains independent queues for each flow, with incoming packets being appended to the *end* of the corresponding flow queue. At any moment, the policy prioritizes the flow whose *leading* (Head-of-Queue) packet has the smallest job-to-packet size ratio; clearly, this maintains the desired in-order property. Then, we can ask: is this WSJF/FCFS hybrid a good policy?

Unfortunately, WSJF-HoQ turns out to be a poor strategy in the adversarial setting. The problem is as follows: while an innocent flow's packets may *typically* have a small job-to-packet size ratio (making this flow a good candidate for service), eventually, a HoQ packet with a large job-to-packet size ratio will stifle the likelihood of the *entire* flow ever being served. Here, the adversary's optimal strategy is simply to send small packets encoding large jobs and wait for this situation to arise.

The fundamental problem with WSJF-HoQ is that *it evaluates entire flows on the basis of one packet, which may not be a good estimator of a flow's candidacy for service.* Based on this observation, we develop another variant of WSJF (hereafter referred to as *WSJF-Inorder*), which predicates its scheduling decision on all queued packets in

the flow queue. As before, the policy maintains independent queues for each flow, with incoming packets appended to the tail of the corresponding queue. In scheduling , the policy computes a *rank* for each flow, f, and prioritizes the flow with the *lowest* rank:

$$Rank(f) = \frac{\Sigma_i J_i(f)}{\Sigma_i P_i(f)},$$

where $J_i(f)$ and $P_i(f)$ denote the job size and packet size of the i'th packet currently in f's flow queue, respectively. Thus, a flow's rank represents its *outstanding work per bit*. In the limit, this converges to $\frac{\mathbf{E}[J(f)]}{\mathbf{E}[P(f)]}$, the long-running average of the flow's inverse-throughput; by minimizing this quantity, WSJF-Inorder maximizes the overall throughput. Consequently, if an adversary wants the policy to consistently schedule (their) large jobs, they must offset the resulting work with a proportionally large number of packet bits, effectively reducing the displacement factor they can achieve. *We use WSJF-Inorder to protect the TCP Reassembly component in Pigasus.*

5.4 Designing Adversary-Proof Schedulers

The final practical issue that we need to address is how to make sure that the scheduler itself will not expose a novel attack surface. While simple policies like FCFS can be implemented with minimal overhead, in order to implement WSJF we must be able to determine which packet has the minimum job-to-packet size ratio on a packet-by-packet basis. If this is done inefficiently, the scheduler itself may become a bottleneck. Another potential problem is that we can only hold a finite number of outstanding packets at any given time. Once the packet buffer becomes full, the system must drop packets in a way that cannot be exploited by an attacker.

There is extensive literature on designing efficient priority queues for packet scheduling in both hardware and software [2, 39, 41, 46, 53]. However, these schedulers typically handle buffer space exhaustion by simply dropping any incoming packet when the buffer is full [46]. While this approach simplifies their design—since they only need to support either `EXTRACT-MIN` or `EXTRACT-MAX` operations, and not both—it does not work well in the adversarial setting. For instance, suppose that we use PIFO [46] to implement WSJF and drop all incoming packets once we run out of buffer space. In this scheme, an attacker can quickly fill up the queue (with minimally-sized packets encoding maximally-sized jobs), eventually leaving the scheduler with no alternative but to pick the attacker's packets. *To avoid this issue, the scheduler must use* `EXTRACT-MIN` *to decide which packet to process next, and* `EXTRACT-MAX` *to decide which packet to drop once it runs out of buffer space.*

We augment the highly-efficient Hierarchical FFS (Find First Set) Queue [39, 53] to provide both `EXTRACT-MIN` and `EXTRACT-MAX` functionality by using a BSF (Bit Scan Forward) instruction to find the minimum element in each bitmap, and a BSR (Bit Scan Reverse) instruction[7] to find the maximum element. Figure 9 depicts the data-structure. An hFFS queue using 32-bit bitmaps and a height of h can represent 32^h unique priorities, and guarantees a worst-case run-time of $O(h)$ (i.e., constant) for all queue operations (`INSERT`, `EXTRACT-MIN`, and `EXTRACT-MAX`).

[7] On modern CPUs, both BSR/BSF translate to single μops with a fixed latency of 3-5 cycles [25].

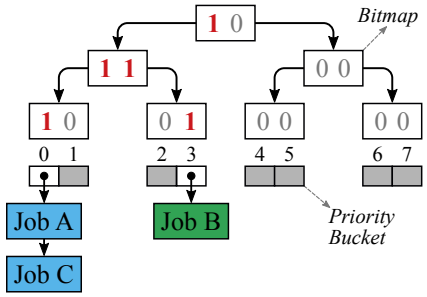


Figure 9: A Hierarchical FFS Queue implemented using 2-bit bitmaps and height of $h = 3$. A '1' in any bitmap indicates a non-empty priority bucket in the subtree rooted at that node. In order to find the *min* (or *max*) priority bucket, we recursively follow the leftmost (or rightmost) set bit.

In order to enable SurgeProtector to work in a general context, we implement the Hierarchical FFS Queue in both hardware and software. In Pigasus, the hardware and software implementations are used to realize WSJF queueing for TCP Reassembly and Full Matching, respectively. The hardware version is implemented in Verilog, operates at 250MHz, and is fully-pipelined, capable of performing one queue operation every FPGA cycle (4 ns). The software version is implemented in C++, and is further evaluated in §6.3.

6 Evaluation

In this section, we evaluate the effectiveness of using SurgeProtector to defend against ACAs on the TCP Reassembler and Full Matching stage of the Pigasus IDS. We also evaluate the robustness of the Hierarchical FFS Queue (used to implement WSJF) against attacks targeting the scheduler itself.

6.1 SurgeProtector + Pigasus

How effective is SurgeProtector at mitigating ACAs on the TCP Reassembler? To answer this question, we emulate an adversary targeting Pigasus' TCP Reassembler using highly out-of-order attack flows, and measure the achieved performance in two modes of operation: using Pigasus' default scheduling policy (FCFS), and using SurgeProtector. For the purpose of this experiment, we use a synthetic trace containing innocent flows sampled from the 2014 CAIDA San Jose dataset [52], and 50 artificially-crafted attack flows.

The attack flows are crafted as follows: we send 1B TCP packets with *alternating sequence numbers* starting with the ISN (i.e., ISN, ISN+2, ISN+4, and so on). With a sequence of N such packets, we can emulate an average adversarial job size corresponding to $\frac{1}{N}\sum_{i=0}^{N-1} i = \frac{(N-1)}{2}$ traversals. We use the optimal adversarial strategy for each mode of operation. In particular, for FCFS, we let N grow to Pigasus' maximum TCP window size of 16KB (by design, Pigasus will drop the flow at this point), then start over. For WSJF, we solve Eq. (9) to determine the optimal adversarial job size, then choose N so as to achieve, on average, the corresponding number of traversals.

Empirically, we find that the maximum serviceable traffic rate of the system (i.e., r_{max}) is 12Gbps, and we fix the input rate for innocent traffic to 10Gbps (corresponding to ~83% load). Figure 10 depicts the steady-state goodput in each mode of operation as we sweep the adversary's attack rate.

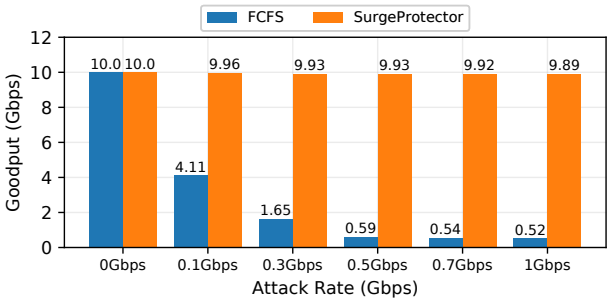


Figure 10: Goodput of Pigasus' TCP Reassembler under FCFS and SurgeProtector.

We observe that the goodput under FCFS drops significantly as the attack rate increases (*e.g.*, with an attack rate of 0.1Gbps, the adversary is able to displace ~5.9Gbps of innocent traffic). Conversely, with SurgeProtector, the goodput remains steady despite the increasing attack rate; in the worst case, at most 0.11Gbps of innocent traffic is displaced.

In lieu of precise knowledge about the system design or the innocent traffic distribution, a practical adversary may also choose to 'probe' the space of attack parameters to determine the most effective adversarial strategy. In order to evaluate performance in this scenario, we emulate an adversary who incrementally changes the degree of out-of-orderness of attack flows while keeping the attack rate fixed at 0.3Gbps. Figure 11 depicts the steady-state TCP Reassembly goodput with FCFS and SurgeProtector as we sweep the out-of-orderness of attack flows (measured in terms of the maximum number of concurrent out-of-order packets within each attack flow).

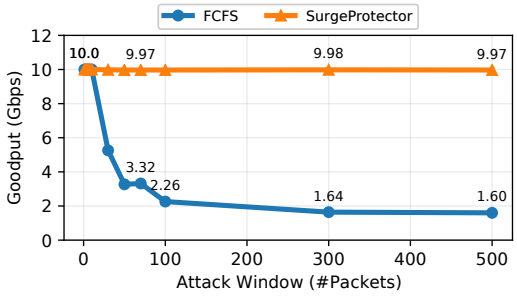


Figure 11: Goodput of Pigasus' TCP Reassembler for different degrees of out-of-orderness of attack flows.

As expected, the goodput under FCFS gradually decreases as the attack flows become increasingly out-of-order (corresponding to larger job sizes per packet), while the goodput under SurgeProtector remains relatively unchanged.

How effective is SurgeProtector at mitigating ACAs on the Full Matching stage? As before, we answer this question by emulating an adversary targeting Pigasus' Full Matching stage, and measure the goodput under FCFS and SurgeProtector. We use a synthetic trace containing innocent flows sampled from all the traces used in [58]. In order to generate attack traffic, we pick the packet payload with the largest job size among all packets in the dataset, and craft an attack flow using this payload for every packet. Figure 12 depicts the steady-state goodput in each mode of operation as we sweep the adversary's attack rate. Once again, we observe that SurgeProtector significantly reduces the impact of the attack

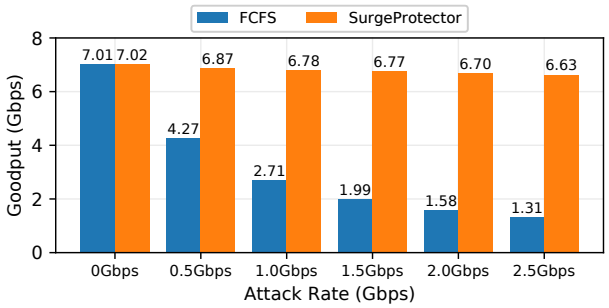


Figure 12: Goodput of Pigasus' Full Matcher under FCFS and SurgeProtector.

on innocent traffic compared to FCFS. In particular, we observe a maximum reduction in goodput of 0.4Gbps for SurgeProtector (compared to 5.7Gbps for FCFS).

6.2 SurgeProtector in Simulation

While the empirical evaluation in §6.1 demonstrates the efficacy of SurgeProtector in the context of a real system, it focuses a small number of attack input rates with just two scheduling policies. In order to analyze a wider range of scheduling policies, applications, and a truly optimal adversary (i.e., one who is not constrained by the space of 'practical' attack strategies[8]), we turn to an adversarial scheduling simulator that we developed in-house. The event-driven simulator, implemented in C++, is capable of modeling G/G/1/k queueing systems, supports both trace-driven and synthetic workloads, and exposes a convenient interface for plugging in a wide range of simulated application backends. An overview of the simulator pipeline is depicted in Figure 13.

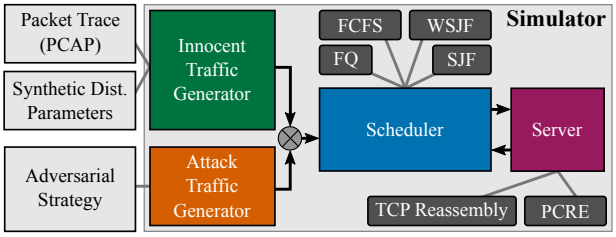


Figure 13: Simulator pipeline.

In order to quickly explore the space of different policies and heuristics for a variety of NFs, the simulator framework allows users to develop and 'run' their own simulated applications on the Server. It also provides traffic-generation modules for innocent and attack traffic, and includes tools for computing the optimal adversarial strategy under SJF and WSJF given innocent job and packet size distributions. The tools numerically solve (8) and (9) to determine the values of J_A and P_A for the given configuration. We use now use the simulator to address several research questions of interest.

[8] In particular, a practical adversary may not be powerful enough to craft packets with a *specific* job size. For instance, in the case of TCP reassembly, an adversary cannot, in practice, force K linked-list traversals on *every* attack packet; instead, they must settle for a uniform distribution over $\{0, \ldots, 2K+1\}$ (see §6.1), resulting in an *average* job size corresponding to K traversals. Simulation allows us to model a more powerful adversary who can precisely control their packets' job sizes.

What is the worst-case DF an optimal adversary can achieve assuming the true job size is known *a priori*? Unlike the empirical setting, the simulator allows us to determine the *true* job size ahead of time. In the following simulated experiments, we use this information for the purpose of scheduling. In the context of TCP Reassembly, Figure 14 depicts the goodput and Displacement Factor achieved by different scheduling policies for various combinations of the input rate (r_I) and attack rate (r_A).[9] Each column corresponds to a certain, fixed r_I (going from 1Gbps on the left, to 5Gbps, and 10Gbps). On the X-axis, we sweep the input attack rate from 10Mbps to 10Gbps. The bottom row depicts the steady-state goodput (in Gbps) as a function of the attack rate, while the top row depicts the corresponding DF.

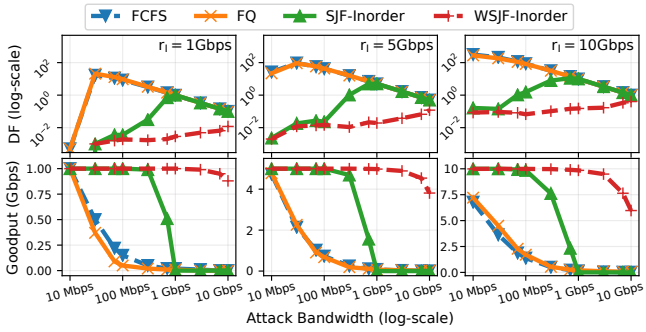


Figure 14: Goodput and Displacement Factor (DF) for TCP Reassembly. Left to right: Increasing innocent input rate, r_I, from 1Gbps to 10Gbps.

Looking at the bottom row, we observe a sharp drop-off in goodput for both FCFS and FQ even with a small attack bandwidth. For instance, with just 30Mbps of attack traffic, an adversary is able to displace roughly half the system goodput, regardless of the innocent input rate. Correspondingly, we see a maximum displacement factor of 313 and 278 for FCFS and FQ, respectively. SJF is initially competitive, but we observe a performance cliff when the attack rate is sufficiently large; with 0.7Gbps of attack traffic, an adversary is able to consistently displace over 50% of the goodput, corresponding to a maximum displacement factor of 11 (recall that the theoretical bound is $\alpha_{SJF} \leq \frac{\mathrm{E}[P]}{P_{min}} \cdot \rho \approx 16$). Finally, we see that WSJF consistently outperforms the other policies, yielding a low degradation in goodput even with a high fraction of attack traffic. We observe a worst-case displacement factor of 0.4 for this application, implying that the *adversary must use over 2.5 bps of their own bandwidth in order to displace 1 bps of innocent traffic*, a considerable improvement over FCFS and FQ.

Similarly, Figure 15 depicts the goodput and DF achieved by the different scheduling policies in the context of Pigasus' Full Matching stage. The format of the figure is identical to that of Figure 14. Looking at the bottom row, we observe a gradual decrease in goodput for FCFS and FQ as the input attack rate increases from 1Mbps to 1Gbps. Overall, we observe a maximum displacement factor of 82 and 75 for FCFS and FQ, respectively. While we don't observe a goodput 'cliff' that we saw for SJF earlier, the adversary is consistently able to displace roughly 50% of the system goodput using an attack

[9]Note that, for each configuration, we use the attack parameters (P_A and J_A) corresponding to the optimal adversarial strategy. For FCFS and FQ, this corresponds to using minimally-sized packets encoding maximally-sized jobs. For SJF and WSJF, we (numerically) solve (8) and (9) to determine these quantities.

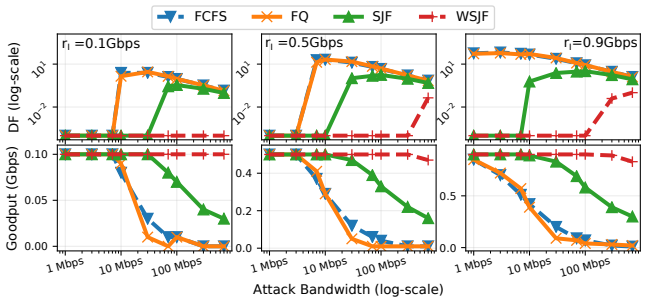


Figure 15: Goodput and Displacement Factor (DF) for Pigasus Full Matching. Left to right: Increasing innocent input rate (r_I) from 0.1Gbps to 0.9Gbps.

bandwidth of 100Mbps, with a maximum observed displacement factor of 3. Finally, WSJF consistently outperforms the other policies, achieving a maximum DF of 0.1.

How does using a heuristic affect the DF achieved by WSJF? In the above simulated experiments, we assumed *a priori* knowledge of a packet's true job size at the time of scheduling. However, given that this information is rarely (if ever) available ahead of time in real systems, we would like to know the impact of using a *heuristic* on the achievable DF. While deriving an analytical answer to this question is beyond the scope of this work, we address it empirically here. For both Pigasus components (TCP Reassembly and Full Matching), we evaluate the difference in DFs achieved under WSJF *with* and *without* their respective heuristics. We assume that the adversary has knowledge of both the actual and estimated job size distributions, and uses job sizes which displace the maximum innocent traffic under the heuristic.[10] While we note that an attacker with such detailed knowledge of the system state likely does not exist, we find that our heuristics perform well even in the face of such an overpowered attacker.

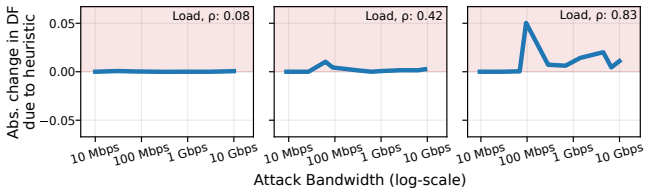


Figure 16: Absolute change in DF achieved by WSJF-Inorder due to the heuristic. Portions highlighted in red indicate regions where the heuristic does worse than the baseline.

In the context of TCP Reassembly, Figure 16 depicts the effect of using the heuristic (described in §5.2) on the achieved DF under WSJF-Inorder. On the x-axis, we sweep the adversary's attack bandwidth (r_A), and on the y-axis we plot the change in DF when using the heuristic (compared to using the true job size, computed offline). We see that using the heuristic increases the DF by at most 0.05 compared to the ideal case. Empirically, we find that this simple heuristic is both an excellent estimator of job size for innocent traffic and largely robust to any subversion attempts by the adversary. We observe similar results (<5% change) for the Full Matching stage.

[10]In practice, this involves a brute-force search over the joint distribution of estimated and actual job sizes for innocent traffic.

6.3 SurgeProtector Scheduler

A key component of the SurgeProtector scheduler is the Hierarchical FFS Queue (§5.4) used to implement WSJF. In this section, we evaluate SurgeProtector against attacks targeting the software heap implementation.

There are two attack vectors we must consider. First, the adversary may flood the fixed-size queue with large attack jobs, causing innocent jobs arriving later to be dropped. Second, the adversary may attempt to inflate their packet arrival rate (using minimally-sized attack packets) beyond what the queue can sustain. Combining these ideas, the adversarial strategy is clear: use minimum-sized packets encoding large jobs.

As basis for this discussion, we consider three WSJF queue designs: a standard, bounded Fibonacci heap that supports `EXTRACT-MIN` operations (but no `EXTRACT-MAX`); a double-ended priority queue [40] (DEPQ, implemented using a pair of Fibonacci heaps) that supports both `EXTRACT-MIN` and `EXTRACT-MAX` operations in worst-case logarithmic time; and finally, the Hierarchical FFS Queue. For the purpose of evaluation, the packet size and job size for innocent traffic are sampled *i.i.d.* from Gaussian distributions (with an average packet size, $\mathbb{E}[P]$, of 1250 bytes, and an average job size, $\mathbb{E}[J]$, of 1μs). We set the maximum job size, J_{max}, to 10μs.

Finally, the experiment setup is as follows. For each of the three heap designs, we pin a process running a software implementation of the heap to a single core on an Intel Xeon E5-2620 CPU operating at 2.1 GHz, where it consumes packets from a 100G Ethernet link via DPDK. The packets (encoding the job size in μs) are dispatched to a different core, which emulates 'running' the job by sleeping for a period corresponding to the job size. A third core is responsible for profiling the application goodput. Figure 17 depicts how the goodput varies with the input attack rate for the three heap implementations.

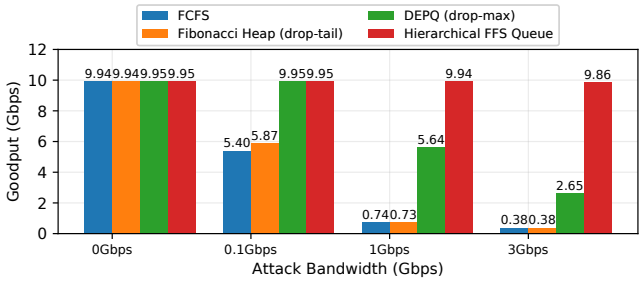


Figure 17: Goodput for different heap implementations.

First, in the case of the standard Fibonacci heap, we observe a large performance cliff when the attack rate reaches a certain threshold. The reason is that, once the queue becomes full, dropping at the tail causes a significant fraction of subsequent innocent arrivals to be dropped. Conversely, while the DEPQ is capable of selectively dropping large jobs, the worst-case logarithmic cost of `EXTRACT-*` operations imposes a significant performance penalty, resulting in a gradual degradation in goodput. Finally, we observe that the Hierarchical FFS Queue's goodput remains largely unchanged regardless of the input attack rate. Overall, we find that the heap's `EXTRACT-MAX` functionality, in conjunction with the worst-case constant complexity of all operations, makes the Hierarchical FFS Queue robust to these kinds of attacks.

7 Limitations and Open Questions

This work opens up a broad range of theoretical and practical questions, and we are only able to answer some of them.

Optimality and Multi-Server Settings: An important theoretical question relates to the existence of an *optimal* adversarial scheduling policy. In this work, we have shown that WSJF, the policy underpinning SurgeProtector, achieves a DF that is always upper-bounded by 1. However, devising a policy that is always optimal (i.e., one which minimizes the DF for any load and choice of traffic parameters) – or proving its existence thereof – remains an open problem. Additionally, we have only considered a queueing system with *one* server; we do not currently know how the ACA mitigation problem scales with more than one server.

Heuristics: As described in §5.2, any practical implementation of SurgeProtector must rely on application-specific heuristic functions for estimating job sizes. Our experience implementing SurgeProtector in the context of TCP reassembly and IDS/IPS Full Matching suggests that even simple, easy-to-compute heuristics can be powerful job sizes estimators. However, the design of heuristics for a broader range of NFs remains an open problem. In particular, there are two questions of interest. First, is there some fundamental property of NFs that makes job size estimation feasible? Second, for NFs in which job size estimation *is* feasible, how do we reason about the efficacy of different heuristic functions? Parallel work in our group [10] has formalized sufficient criteria for an 'ideal' heuristic, and has shown that non-ideal heuristics can still provide an upper-bound on the DF achievable under WSJF.

Preemption: In this work, we have only explored the space of *non-preemptive* scheduling (i.e., a job, once started, must run to completion). However, given recent advances in the design and implementation of lightweight preemption handlers [7], it is reasonable to ask: can we do even better with preemptive scheduling policies? This is particularly relevant for NFs where developing accurate heuristics is challenging. In this case, preemption may help tolerate some error in job size estimates by allowing the scheduler an additional degree of freedom (*e.g.*, by preempting jobs that far exceed their job size estimates).

Fairness: As we have seen in §4.2, fair queueing is fundamentally vulnerable to ACAs because of the adversary's ability to spawn many flows. However, fairness is an important consideration for many NFs. While WSJF alone does not provide any fairness guarantees, we conjecture that an augmentation of this policy (*e.g.*, using FQ as a second-stage queueing discipline, or switching between the two based on some goodput watermark) may be able to provide both ACA resilience and flow-level fairness.

Memory Complexity Attacks: Finally, we have not considered the impact of ACAs on *memory*. In many systems, memory is just as precious (and exhaustible) a resource as processing cycles, and may be an important consideration in the design and analysis of adversarial scheduling policies for NFs.

8 Related Work

ACAs and mitigation: Crosby et al. were the first to characterize ACAs as Denial-of-Service (DoS) vectors in [12], and empirically evaluated their impact in the context of an IDS. Others have

since explored ACAs on a variety of applications, including hash tables [3, 4], automata-based multi-string pattern-matching [43], regular expression matching [16, 42, 56], PDF decompression [26], and TCP reassembly [18, 50]. [36] provides both an excellent survey of prior work and a novel approach for automatically crafting ACAs in a domain-independent manner (using fuzzing).

Many works have proposed *application-specific* mitigation strategies. For example, [44] implements TCP reassembly by maintaining statically-allocated, fixed-sized buffers for each flow; this renders the design impervious to ACAs at the cost of significantly higher memory overhead (every flow is allocated 64KB of memory regardless of its peak usage). Similarly, many regular expression engines restrict the number of states a single packet may invoke to avoid ReDoS attacks [48] (limiting the length of regular expressions in the common case). Other systems place a cap on the number of cycles spent decompressing a file or webpage for deep packet inspection [24] (limiting the size of files or web pages that can be served). Still other systems rely on universal hashing to prevent attacks on hash tables [12] (imposing computational and memory overheads). In a slightly different direction, [1] leverages a multi-core architecture to mitigate ACAs on DPI engines.

Scheduling: Scheduling and queueing theory has garnered significant research attention in recent years. While the vast majority of queueing literature focuses on optimizing various response time metrics in stochastic settings, some recent works in OS and packet scheduling are notable due to the focus on fairness and performance isolation. In particular, Fair Queueing (FQ) [17] aims to equitably partition the available link bandwidth between multiple contending flows. Dominant Resource Fair Queueing (DRFQ) [21] generalizes this idea to multiple resources [22], and [54] provides a low-overhead approximation to DRFQ. However, as described in §4.2, FQ and its variants are ineffective in the adversarial setting [57].

Recent works have also explored the use of queueing theory to analyze *volumetric* DoS attacks (*e.g.*, SYN-floods). [55] proposes a two-dimensional embedded Markov chain to model DoS attacks, and derives various performance metrics (*e.g.*, connection loss probability) by analyzing its stationary distribution. Along these lines, [6] evaluates how dynamic TCP timeouts can be used as a mitigation strategy against SYN-floods. [37] proposes a composite model to jointly analyze memory and bandwidth resource exhaustion during an attack. More recently, [20] derived the feasibility criteria for a successful volume-based DDoS attack on a multi-hop network following the Join-the-Shortest-Queue (JSQ) policy. We reiterate that the distinguishing factor here is the *type* of DoS attack considered in this work: complexity-based instead of volumetric.

Finally, we are aware of two works that consider the ACA mitigation problem from a queueing theoretic perspective, and are therefore most closely related to this work. First, [28] models DoS attacks using an M/M/1/k queueing model with the goal of detecting both flood- and complexity-based attacks. However, they only perform analysis for FCFS, and they only consider exponentially-distributed service times (which may not be an accurate assumption in the adversarial setting). Second, [4] analyzes the impact of using two different hashing schemes on the efficacy of ACAs on hash tables. They also develop a metric called the 'Vulnerability Factor' to quantify the impact of ACAs. However, they limit their analysis to FCFS. Moreover, since their analysis is based on a job's *average waiting time*, they are fundamentally constrained to scenarios where the system is *not* overloaded.

To the best of our knowledge, this is the first work to analyze scheduling policies beyond the simple FCFS and to propose a policy-based mitigation strategy for ACAs.

9 Conclusion

Network functions on the Internet are prone to algorithmic complexity attacks (ACAs), a potent class of Denial-of-Service (DoS) attacks. We designed SurgeProtector, a framework to mitigate temporal ACAs on NFs using novel insights from adversarial scheduling theory. SurgeProtector provides provable upper bounds on the maximum 'harm per unit effort' an adversary can induce, regardless of the underlying NF application, the system load, and parameters of the innocent traffic distribution. Our proofs and evaluation show that WSJF, the scheduling algorithm behind SurgeProtector, provides resilience to ACAs without limiting the underlying algorithms in the NF.

10 Acknowledgements

We thank our shepherd, Alan Liu, and the anonymous reviewers for their insightful comments. We also thank Jalani Williams and Isaac Grosof for helpful discussions regarding the underlying theory, Zhipeng Zhao and Siddharth Sahay for their help navigating the Pigasus source code, and Vyas Sekar for his feedback on an early draft of this paper. We are also grateful to the Parallel Data Lab (PDL) at CMU for providing compute resources to us. This work was funded by Intel and VMware through the Intel/VMware Crossroads 3D-FPGA Academic Research Center, a VMWare Systems Research Award, a Cylab Presidential Fellowship, and a Google Research Gift.

This work does not raise any ethical concerns.

References

[1] Yehuda Afek, Anat Bremler-Barr, Yotam Harchol, David Hay, and Yaron Koral. MCA2: Multi-Core Architecture for Mitigating Complexity Attacks. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, page 235–246, New York, NY, USA, 2012. Association for Computing Machinery.

[2] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-in First-out Behaviors Using Strict-Priority Queues. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI '20, page 59–76, USA, 2020. USENIX Association.

[3] Noa Bar-Yosef and Avishai Wool. Remote Algorithmic Complexity Attacks against Randomized Hash Tables. In Joaquim Filipe and Mohammad S. Obaidat, editors, *E-business and Telecommunications*, pages 162–174, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[4] Udi Ben-Porat, Anat Bremler-Barr, and Hanoch Levy. Vulnerability of Network Mechanisms to Sophisticated DDoS Attacks. *IEEE Transactions on Computers*, 62(5):1031–1043, 2013.

[5] Ethan Blanton and Mark Allman. On Making TCP More Robust to Packet Reordering. *SIGCOMM Comput. Commun. Rev.*, 32(1):20–30, January 2002.

[6] Daniel Boteanu and José M. Fernandez. A Comprehensive Study of Queue Management as a DoS Counter-Measure. *Int. J. Inf. Secur.*, 12(5):347–382, October 2013.

[7] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Lightweight preemptible functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 465–477, 2020.

[8] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The battle of the schedulers: Freebsd ULE vs. linux CFS. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 85–96, Boston, MA, July 2018. USENIX Association.

[9] Ben Caller. Regexploit: Dos-able regular expressions, Mar 2021.

[10] Erica Chiang, Nirav Atre, and Hugo Sadok. Robust Heuristics: Attacks and Defenses for Job Size Estimation in WSJF Systems. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22 Demos and Posters)*. Association for Computing Machinery, August 2022.

[11] Alan Cobham. Priority Assignment in Waiting Line Problems. *Journal of the Operations Research Society of America*, 2(1):70–76, 1954.

[12] Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.

[13] Levente Csikor, Dinil Mon Divakaran, Min Suk Kang, Attila Kőrösi, Balázs Sonkoly, Dávid Haja, Dimitrios P. Pezaros, Stefan Schmid, and Gábor Rétvári. Tuple Space Explosion: A Denial-of-Service Attack against a Software Packet Classifier. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, page 292–304, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Levente Csikor, Vipul Ujawane, and Dinil Mon Divakaran. On the Feasibility and Enhancement of the Tuple Space Explosion Attack against Open vSwitch, 2020.

[15] Levente Csikor, Vipul Ujawane, and Dinil Mon Divakaran. On the Feasibility and Enhancement of the Tuple Space Explosion Attack against Open vSwitch, 2020.

[16] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 246–256, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, August 1989.

[18] Sarang Dharmapurikar and Vern Paxson. Robust TCP Stream Reassembly in the Presence of Adversaries. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM '05, page 5, USA, 2005. USENIX Association.

[19] Eric Dumazet. Merge Branch 'tcp-robust-ooo'. https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net.git/commit/?id=1a4f14bab1868b443f0dd3c55b689a478f82e72e, 2018.

[20] Xinzhe Fu and Eytan Modiano. Fundamental Limits of Volume-Based Network DoS Attacks. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(3), December 2019.

[21] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-Resource Fair Queueing for Packet Processing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, page 1–12, New York, NY, USA, 2012.

[22] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11, page 323–336, USA, 2011. USENIX Association.

[23] Gaston H Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM (JACM)*, 28(2):289–304, 1981.

[24] Google. Google Drive. https://support.google.com/a/answer/172541, 2021.

[25] Torbjörn Granlund. Instruction latencies and throughput for amd and intel x86 processors. *Technical report, KTH*, 2012.

[26] Nathan Hauke and David Renardy. Denial of Service with a Fistful of Packets: Exploiting Algorithmic Complexity Vulnerabilities. https://www.blackhat.com/us-19/briefings/schedule/#denial-of-service-with-a-fistful-of-packets-exploiting-algorithmic-complexity-vulnerabilities-16445, 2019.

[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[28] Suraiya Khan and Issa Traore. Queue-based Analysis of DoS Attacks. In *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, pages 266–273, 2005.

[29] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 254–265, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Ka-Cheong Leung, Victor O. K. Li, and Daiqin Yang. An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges. *IEEE Trans. Parallel Distrib. Syst.*, 18(4):522–535, April 2007.

[31] Yin Li, Chuang Lin, Fengyuan Ren, and Yifeng Geng. H-PFSP: Efficient Hybrid Parallel PFSP Protected Scheduling for MapReduce System. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1099–1106, 2013.

[32] Linux Kernel. CFS Scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html, 2021.

[33] Yang Liu, Yukun Zeng, and Xuefeng Piao. High-Responsive Scheduling with MapReduce Performance Prediction on Hadoop YARN. In *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 238–247, 2016.

[34] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, Santa Clara, CA, February 2020. USENIX Association.

[35] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. Automated Synthesis of Adversarial Workloads for Network Functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 372–385, New York, NY, USA, 2018.

[36] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2155–2168, New York, NY, USA, 2017.

[37] Simona Ramanauskaitė and Antanas Čenys. Composite DoS Attack Model/Jungtinis DoS Atakų Modelis. *Mokslas–Lietuvos ateitis/Science–Future of Lithuania*, 4(1):20–26, 2012.

[38] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, page 229–238, USA, 1999. USENIX Association.

[39] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. Eiffel: Efficient and flexible software packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 17–32, Boston, MA, February 2019. USENIX Association.

[40] Sartaj Sahni. Double-ended priority queues. In *Handbook of Data Structures and Applications*, 2004.

[41] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, Santa Clara, CA, February 2020. USENIX Association.

[42] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. ReScue: Crafting Regular Expression DoS Attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 225–235, New York, NY, USA, 2018. Association for Computing Machinery.

[43] Govind Sreekar Shenoy, Jordi Tubella, and Antonio González. Improving the Resilience of an IDS against Performance Throttling Attacks. In Angelos D. Keromytis and Roberto Di Pietro, editors, *Security and Privacy in Communication Networks*, pages 167–184, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[44] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43. IEEE, 2015.

[45] Shan Sinha, Srikanth Kandula, and Dina Katabi. Harnessing TCP's Burstiness with Flowlet Switching. In *Proc. 3rd ACM Workshop on Hot Topics in Networks (Hotnets-III)*. Citeseer, 2004.

[46] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 44–57, New York, NY, USA, 2016. Association for Computing Machinery.

[47] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking Algorithmic Complexity Attacks against a NIDS. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 89–98, 2006.

[48] Snort Project. SNORT Users Manual. https://www.snort.org/documents/snort-users-manual, 2020.

[49] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, page 135–146, New York, NY, USA, 1999. Association for Computing Machinery.

[50] Juha-Matti Tilli. CVE-2018-5390: Linux Kernel TCP Reassembly Algorithm Lets Remote Users Consume Excessive CPU Resources on the Target System. https://ubuntu.com/security/cve-2018-5390, 2018.

[51] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[52] Colby Walsworth, Emile Aben, K Claffy, and D Andersen. The caida anonymized 2019 internet traces, 2019.

[53] Hao Wang and Bill Lin. Per-flow queue management with succinct priority indexing structures for high speed packet scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1380–1389, 2013.

[54] Wei Wang, Ben Liang, and Baochun Li. Low Complexity Multi-Resource Fair Queueing with Bounded Delay. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 1914–1922, 2014.

[55] Yang Wang, Chuang Lin, Quan-Lin Li, and Yuguang Fang. A Queueing Analysis for the Denial of Service (DoS) Attacks in Computer Networks. *Comput. Netw.*, 51(12):3564–3573, August 2007.
[56] Valentin Wüstholz, Oswaldo Olivo, Marijn J. Heule, and Isil Dillig. Static Detection of DoS Vulnerabilities in Programs That Use Regular Expressions. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*, page 3–20, Berlin, Heidelberg, 2017. Springer-Verlag.
[57] Xiaowei Yang, David Wetherall, and Thomas Anderson. A DoS-limiting network architecture. *ACM SIGCOMM Computer Communication Review*, 35(4):241–252, 2005.
[58] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1083–1100. USENIX Association, November 2020.

A Proofs for DF Analysis

A.1 Proof of Claim 1 (DF of FCFS)

Proof. Consider any innocent input traffic rate r_I and any packet size and job size distributions with expectations $\mathbb{E}[P]$ and $\mathbb{E}[J]$. Observe that, over any time period of length T seconds, the number of *class I* packets appearing on the ingress link is $N_I = \frac{r_I T}{\mathbb{E}[P]}$. Similarly, the number of *class A* packets appearing on the link over the same period is $N_A = \frac{r_A T}{P_{\min}}$. FCFS guarantees that these $(N_I + N_A)$ jobs will be scheduled before any jobs that arrive afterwards. Also, the total time required to serve these jobs is $(N_I \cdot \mathbb{E}[J] + N_A \cdot J_{\max})$ seconds, yielding in expectation $N_I \cdot \mathbb{E}[P]$ bits worth of innocent traffic on the egress link. Thus, in the long-run, the goodput o_I can be upper-bounded as follows:

$$o_I(r_I, r_A) \le \lim_{T\to\infty} \frac{N_I \cdot \mathbb{E}[P]}{N_I \cdot \mathbb{E}[J] + N_A \cdot J_{\max}} = \frac{r_I}{r_I \cdot \frac{\mathbb{E}[J]}{\mathbb{E}[P]} + r_A \cdot \frac{J_{\max}}{P_{\min}}}.$$

We can then lower-bound the DF $\alpha_{\text{FCFS}}(r_I)$ as follows:

$$\alpha_{\text{FCFS}}(r_I) = \sup_{r_A} \frac{\min\{r_I, r_{\max}\} - o_I(r_I, r_A)}{r_A} \tag{2}$$

$$\ge \sup_{r_A} \frac{1}{r_A} \left(\min\{r_I, r_{\max}\} - \frac{r_I}{r_I \cdot \frac{\mathbb{E}[J]}{\mathbb{E}[P]} + r_A \cdot \frac{J_{\max}}{P_{\min}}} \right) \tag{3}$$

$$\ge \frac{J_{\max}}{P_{\min}} \frac{\min\{r_I, r_{\max}\}}{2 r_I \left(\frac{2}{\min\{r_I, r_{\max}\}} - \frac{1}{r_{\max}} \right)}, \tag{4}$$

where recall that $r_{\max} = \frac{\mathbb{E}[P]}{\mathbb{E}[J]}$, (2) is true since the goodput is $\min\{r_I, r_{\max}\}$ under FCFS in the absence of adversarial traffic, (3) applies the upper bound on $o_I(r_I, r_A)$, and (4) is obtained by setting r_A as follows: $r_A = \frac{r_I P_{\min}}{J_{\max}} \left(\frac{2}{\min\{r_I, r_{\max}\}} - \frac{1}{r_{\max}} \right)$. Therefore, $\alpha_{\text{FCFS}}(r_I) \to +\infty$ as $\frac{J_{\max}}{P_{\min}} \to +\infty$. □

A.2 Proof of Claim 2 (DF of FQ)

Proof. Assume that the input traffic rate for innocent traffic, r_I, is split equally among k innocent flows, while each packet of adversarial traffic corresponds to a distinct attack flow. As in FCFS, the adversary maximizes the harm to the system by crafting packets with the smallest possible packet size $P_{\min}$ and the largest possible job size $J_{\max}$.

Consider the state of the system at time $T > J_{\max}$. Observe that, in expectation, the *maximum number* of innocent jobs in each of the k flow queues with virtual clock $\le T$ is $N_I = \frac{T}{\mathbb{E}[J]}$. Conversely, the number of adversarial jobs with virtual clock $\le T$ is given by $N_A = \frac{(T - J_{\max}) r_A}{P_{\min}}$. FQ ensures that all $(N_A + k \cdot N_I)$ jobs will be scheduled before any jobs that arrive afterwards. Also, the total time required to serve these jobs is given by the expression: $\frac{(T - J_{\max}) r_A}{P_{\min}} \cdot J_{\max} + k \cdot T$. Then, the goodput o_I can be upper-bounded as follows:

$$o_I(r_I, r_A) \le \lim_{T\to\infty} \frac{k \cdot \frac{T}{\mathbb{E}[J]} \cdot \mathbb{E}[P]}{\frac{(T - J_{\max}) r_A}{P_{\min}} \cdot J_{\max} + k \cdot T} = \frac{k \cdot r_{\max}}{r_A \cdot \frac{J_{\max}}{P_{\min}} + k},$$

where recall that $r_{\max} = \frac{\mathbb{E}[P]}{\mathbb{E}[J]}$. We can then lower-bound the DF $\alpha_{\text{FQ}}(r_I)$ as follows:

$$\alpha_{\text{FQ}}(r_I) = \sup_{r_A} \frac{\min\{r_I, r_{\max}\} - o_I(r_I, r_A)}{r_A} \tag{5}$$

$$\ge \sup_{r_A} \left(\min\{r_I, r_{\max}\} - \frac{k \cdot r_{\max}}{r_A \cdot \frac{J_{\max}}{P_{\min}} + k} \right) \tag{6}$$

$$\ge \frac{J_{\max}}{P_{\min}} \frac{\min\{r_I, r_{\max}\}}{2 k r_{\max} \left(\frac{2}{\min\{r_I, r_{\max}\}} - \frac{1}{r_{\max}} \right)}, \tag{7}$$

where (5) is true since the goodput is $\min\{r_I, r_{\max}\}$ under FQ in the absence of adversarial traffic, (6) applies the upper bound on $o_I(r_I, r_A)$, and (7) is obtained by setting $r_A = \frac{k r_{\max} P_{\min}}{J_{\max}} \left(\frac{2}{\min\{r_I, r_{\max}\}} - \frac{1}{r_{\max}} \right)$. Therefore, $\alpha_{\text{FQ}}(r_I) \to +\infty$ as $\frac{J_{\max}}{P_{\min}} \to +\infty$. □

A.3 Proof of Theorem 1 (DF of SJF)

Optimal attack strategy: We first characterize the optimal attack strategy of the adversary under SJF for a given innocent input traffic rate r_I and a given adversarial input traffic rate r_A. It is easy to see that the adversary should craft packets with the smallest possible packet size $P_{\min}$ since the job scheduling under SJF does not depend on packet sizes.

To reason about the optimal choice of adversarial job sizes, we first consider the case where the adversary picks certain job size J_A. Then innocent jobs with size $\le J_A$ and adversarial jobs have priority over innocent jobs with size $> J_A$. Therefore, innocent jobs with size $> J_A$ will be 'displaced', i.e., never get served, if r_A is large enough to overload the processor with innocent jobs with size $\le J_A$ and adversarial jobs. Consequently, the goodput consists of innocent packets whose job sizes are no larger than J_A.

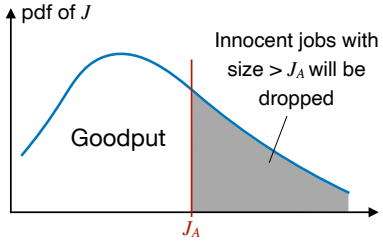


Figure 18: Optimal choice of adversarial job size J_A.

We now argue that the adversary only needs to pick one deterministic job size without loss of optimality. To see this, suppose the adversary crafts packets whose job sizes are either J_A or J'_A, where

$J_A < J'_A$. But if the adversary swaps the packets with job size J'_A for packets with job size J_A, they can only displace more or equal innocent traffic. Therefore, we can restrict our attention to attack strategies with one deterministic job size.

We characterize the adversary's optimal choice for the job size in Lemma 1 below. Here for simplicity, we assume that the innocent packet size P and job size J are independent. We will remove this assumption when we present WSJF for SurgeProtector. Recall that the pdf of the innocent job size J is denoted by $f_J(\cdot)$.

Lemma 1 (**Optimal Attack Strategy for SJF**). *Consider the SJF policy for job scheduling and any innocent input traffic rate r_I and any adversarial input traffic rate r_A. Then the adversary can minimize the goodput by choosing the job size, J_A, to be the solution of the following equation if the solution satisfies $J_A \le J_{\max}$:*

$$\frac{r_I}{\mathbb{E}[P]}\int_0^{J_A} j\cdot f_J(j)\,dj + \frac{r_A}{P_{\min}}\cdot J_A = 1, \tag{8}$$

and $J_A = J_{\max}$ otherwise.

Proof. We have argued that the adversary only needs to pick one deterministic job size. It remains to show that the r_A given in the lemma minimizes the goodput. In (8), if the solution satisfies $J_A \le J_{\max}$, the term $\frac{r_I}{\mathbb{E}[P]}\int_0^{J_A} j\cdot f_J(j)dj$ is the workload for the processor contributed by innocent packets with job size $\le J_A$. Since these packets get served by the processor, they constitute the goodput. We consider the following two cases: (i) *The adversary picks a job size larger than J_A*. In this case, more innocent jobs will get served since smaller jobs are prioritized, resulting in a larger goodput. (ii) *The adversary picks a job size $J'_A < J_A$*. In this case, the total workload from innocent jobs with size $\le J'_A$ and adversarial jobs is given by

$$\frac{r_I}{\mathbb{E}[P]}\int_0^{J'_A} j\cdot f_J(j)dj + \frac{r_A}{P_{\min}}\cdot J'_A < 1.$$

So some innocent jobs with size $> J'_A$ will also get served. More precisely, the processor has more capacity left for innocent jobs when the adversarial job size is J'_A compared to when the adversarial job size is J_A. Thus the goodput is higher under J'_A. Combining the two cases, it follows that the solution J_A to (8) is the optimal choice for the adversary.

When the solution to (8) satisfies $J_A > J_{\max}$, the adversary cannot displace any innocent traffic no matter what the job size is. So simply setting $J_A = J_{\max}$ is an optimal choice. □

The remainder of the proof for Theorem 1 is very similar to that for Theorem 2 in the next section. As such, we elide this part of the proof for the sake of brevity.

A.4 Proof of Theorem 2 (DF of WSJF)

Optimal attack strategy: We again first characterize the optimal attack strategy of the adversary under WSJF for a given innocent input traffic rate r_I and a given adversarial input traffic rate r_A. Under WSJF, the harm that an adversary can induce is fully determined by the job-to-packet-size ratio of the adversarial traffic, denoted as Z_A, as opposed to the individual values of job size and packet size. To see this, note that WSJF schedules jobs solely based on their job-to-packet-size ratios, and that the rate at which the adversary generates work for the processor is $r_A \cdot Z_A$, which also depends on the job size and packet size only through their ratio. Therefore, we assume that the adversary uses packet size $P_{\min}$ without loss of optimality, and picks a job-to-packet-size ratio Z_A that results in job size $Z_A \cdot P_{\min}$.

The reasoning for the optimal choice of Z_A is similar to that for the optimal choice of J_A under SJF. The only difference is that under WSJF, whether an innocent packet gets displaced or not is determined by its job-to-packet-size ratio rather than its job size. Following similar arguments, we establish Lemma 2 below, whose proof is omitted for the sake of brevity. Here we use $f_{P,J}(p,j)$ to denote the joint pdf of the innocent packet size and job size. Note that we do not make independence assumptions between them.

Lemma 2 (**Optimal Attack Strategy for WSJF**). *Consider the WSJF policy for job scheduling and any innocent input traffic rate r_I and any adversarial input traffic rate r_A. Then the adversary can minimize the goodput by choosing the job-to-packet-size ratio, Z_A, to be the solution of the following equation if the solution satisfies $Z_A \cdot P_{\min} \le J_{\max}$:*

$$\frac{r_I}{\mathbb{E}[P]}\int_{P_{\min}}^{P_{\max}}\int_0^{p\cdot Z_A} j\cdot f_{P,J}(p,j)\,dj\,dp + r_A\cdot Z_A = 1, \tag{9}$$

and $Z_A = \frac{J_{\max}}{P_{\min}}$ otherwise.

DF analysis: We now formally prove the upper bound on the DF of WSJF below.

Proof. We divide the discussion into two cases: $r_I < r_{\max}$ (underloaded by innocent traffic) and $r_I \ge r_{\max}$ (overloaded by innocent traffic).

Case 1 ($r_I < r_{\max}$): Consider a period of T seconds, with a total of N innocent packets arriving during this period. Let $S = \{(p_1,j_1),(p_2,j_2),\ldots,(p_N,j_N)\}$ denote this set of arrivals, where $p_i \in [P_{min}, P_{max}]$ and $j_i \in [0, J_{max}]$ denote the packet size and job size corresponding to the i'th packet, respectively. Without loss of generality, we choose the index of each packet, i, such that $\frac{j_i}{p_i} \le \frac{j_{i+1}}{p_{i+1}}\ \forall i$.

We now turn to the service order of these N innocent packets under WSJF. In particular, note that since WSJF serves packets in *increasing order of their job-size-to-packet-size-ratio*, packet *1* is served before packet *2*, packet *2* before packet *3*, and so on. Further, since we assumed that $r_I < r_{max}$, it follows that in steady state (i.e., for sufficiently large T), all N jobs will be served. Now, consider an adversary who wishes to displace $k \in \{1, \ldots, N\}$ innocent packets. In order to do this, they must inject some $x \ge 0$ attack packets with packet size p_A and job size j_A. Note that the attacker's input traffic rate can be written as: $r_A = \lim_{T\to\infty}\frac{x\cdot p_A}{T}$. Now, in order to both be served *and* displace k innocent packets, x, p_A, and j_A must satisfy the following constraints with probability 1:

$$\frac{j_A}{p_A} \le \frac{j_{N-k+1}}{p_{N-k+1}}, \tag{10}$$

$$\sum_{i=1}^{N-k} j_i + x\cdot j_A \ge T - o(T), \tag{11}$$

where (11) further implies that

$$x \ge \frac{1}{j_A}\left(T - o(T) - \sum_{i=1}^{N-k} j_i\right). \tag{12}$$

In particular, (12) ensures that the adversarial workload pushes the system to capacity (otherwise, this slack would be applied towards serving additional traffic, implying that the adversary would fail to displace k innocent packets). Similarly, (10) ensures that all x adversarial packets are served *before* packets $\{N-k+1, ..., N\}$ (otherwise, some of the last k innocent packets would be prioritized over the adversary's traffic).

Let g denote the *number of innocent bits displaced* by the adversary using $x \cdot p_A$ bits of their own traffic. We have: $g(k) = \sum_{i=N-k+1}^{N} p_i$. Now, in steady state, the displacement factor under WSJF can be expressed as follows with probability 1:

$$\begin{aligned}
\alpha_{\text{WSJF}}(r_I, r_A) &= \frac{r_I - o_I(r_I, r_A)}{r_A} \\
&= \lim_{T\to\infty} \frac{g(k)}{x \cdot p_A} \\
&= \lim_{T\to\infty} \frac{\sum_{i=N-k+1}^{N} p_i}{x \cdot p_A} && (13) \\
&\le \lim_{T\to\infty} \frac{\sum_{i=N-k+1}^{N} p_i}{T - \sum_{i=1}^{N-k} j_i} \cdot \frac{j_A}{p_A} && (14) \\
&\le \lim_{T\to\infty} \frac{\overbrace{\sum_{i=N-k+1}^{N} p_i \cdot \frac{j_{N-k+1}}{p_{N-k+1}}}^{\text{Term (R1)}}}{T - \sum_{i=1}^{N-k} j_i}, && (15)
\end{aligned}$$

where (14) is obtained by substituting the expression for x we derived in (12) into (13), and (15) is obtained by substituting the expression for $\frac{j_A}{p_A}$ we derived in (10) into (14). Now, since $\frac{j_i}{p_i} \le \frac{j_{i+1}}{p_{i+1}}$ implying that $p_{i+1} \cdot \frac{j_i}{p_i} \le j_{i+1}\ \forall i$, we can upper-bound Term (R1) as follows:

$$\begin{aligned}
\sum_{i=N-k+1}^{N} p_i \cdot \frac{j_{N-k+1}}{p_{N-k+1}} &\le \sum_{i=N-k+1}^{N} j_i \\
&\le \sum_{i=1}^{N} j_i - \sum_{i=1}^{N-k} j_i.
\end{aligned}$$

Substituting this expression back into (15), we have:

$$\alpha_{\text{WSJF}}(r_I, r_A) \le \lim_{T\to\infty} \frac{\sum_{i=1}^{N} j_i - \sum_{i=1}^{N-k} j_i}{T - \sum_{i=1}^{N-k} j_i}.$$

Observe that the RHS is of the form $h(x) = \frac{t-x}{T-x}$, where $t = \sum_{i=1}^{N} j_i \le T$ (i.e., the cumulative service time for innocent packets in the absence of adversarial traffic, which is constant for a given r_I), and $x = \sum_{i=1}^{N-k} j_i \in [0,\ t]$. Since h is a decreasing function of x on its domain, it follows that $\alpha_{\text{WSJF}}(r_I, r_A)$ achieves its maximum value when $x = 0$. Therefore, we can write:

$$\alpha_{\text{WSJF}}(r_I) \le \lim_{T\to\infty} h(0) = \lim_{T\to\infty} \frac{t}{T} \le \lim_{T\to\infty} \frac{1}{T} \sum_{i=1}^{N} j_i. \qquad (16)$$

Now, for a given distribution of innocent packets and job sizes, the input rate and maximum serviceable rate for innocent traffic (r_I and r_{max}, respectively), can be expressed as follows:

$$r_I = \lim_{T\to\infty} \frac{1}{T} \sum_{i=1}^{N} p_i, \text{ w.p.1,} \qquad (17)$$

$$r_{max} = \frac{\mathbb{E}[P]}{\mathbb{E}[J]} = \lim_{T\to\infty} \frac{\sum_{i=1}^{N} p_i}{N} \frac{1}{\frac{\sum_{i=1}^{N} j_i}{N}} = \lim_{T\to\infty} \frac{\sum_i p_i}{\sum_i j_i}, \text{ w.p.1.} \qquad (18)$$

Then, we can define the *load* on the system due to innocent traffic, ρ, as follows:

$$\rho(r_I) = \frac{r_I}{r_{max}} = \lim_{T\to\infty} \frac{1}{T} \sum_{i=1}^{N} j_i \le 1, \text{ w.p.1.} \qquad (19)$$

Observe that (19) is identical to the RHS of (16). Thus, we can rewrite the maximum DF under WSJF: $\alpha_{\text{WSJF}}(r_I) \le \rho$, as required.

Case 2 ($r_I \ge r_{\max}$): In this case, one can verify that there exists $K < N$ such that the system is underloaded with respect to packets with a job-size-to-packet-size-ratio of $\frac{j_K}{p_K}$ (i.e., the distribution of innocent traffic served is effectively truncated at this point). Following the same arguments as of those for Case 1 (the worst case being $r_I = r_{max}$, corresponding to $\rho = 1$), we have:

$$\alpha_{\text{WSJF}}(r_I) \le \rho.$$

Combining Case 1 and Case 2 completes the proof of Theorem 2. □