A NOVEL PUZZLE-BASED FRAMEWORK FOR MITIGATING DISTRIBUTED DENIAL OF SERVICE ATTACKS AGAINST INTERNET APPLICATIONS

by

Mehmud Abliz

Bachelor of Science, Jilin Univeristy, 2004

Master of Science, University of Pittsburgh, 2011

Submitted to the Graduate Faculty of

the Dietrich School of Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2015

UNIVERSITY OF PITTSBURGH

DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Mehmud Abliz

It was defended on

April 15, 2015

and approved by

Taieb F. Znati, Department of Computer Science

Rami Melhem, Department of Computer Science

Youtao Zhang, Department of Computer Science

Prashant Krishnamurthy, School of Information Sciences

Dissertation Director: Taieb F. Znati, Department of Computer Science

A NOVEL PUZZLE-BASED FRAMEWORK FOR MITIGATING DISTRIBUTED DENIAL OF SERVICE ATTACKS AGAINST INTERNET APPLICATIONS

Mehmud Abliz, PhD

University of Pittsburgh, 2015

Cryptographic puzzles are promising techniques for mitigating DDoS attacks via decreasing the incoming rate of service eligible requests. However, existing cryptographic puzzle techniques have several shortcomings that make them less appealing as a tool of choice for DDoS defense. These shortcomings include: (1) the lack of accurate models for dynamically determining puzzle hardness; (2) the lack of an efficient and effective counter mechanism for puzzle solution replay attacks; and (3) the wastefulness of the puzzle computations in terms of the clients' computational resources. In this thesis, we provide a puzzle based DDoS defense framework that addresses these shortcomings.

Our puzzle framework includes three novel puzzle mechanisms. The first mechanism, called **Puzzle+**, provides a mathematical model of per-request puzzle hardness. Through extensive experimental study, we show that this model optimizes the effectiveness of puzzle based DDoS mitigation while enabling tight control over the server utilization. In addition, Puzzle+ disables puzzle solution replay attacks by utilizing a novel cache algorithm to detect replays.

The second puzzle mechanism, called **Productive Puzzles**, alleviates the wastefulness of computational puzzles by transforming the puzzle computations into computations of meaningful tasks that provide utility. Our third puzzle mechanism, called **Guided Tour Puzzles**, eliminates the wasteful puzzle computations all together, and adopts a novel delay-based puzzle construction idea. In addition, it is not affected by the disparity in the com-

putational resources of the client machines that perform the puzzle computations. Through measurement analysis on real network testbeds as well as extensive simulation study, we show that both Productive Puzzles and Guided Tour Puzzles achieve effective mitigation of DDoS attacks while satisfying no wasteful computation requirement.

Lastly, we introduce a novel queue management algorithm, called **Stochastic Fair Drop Queue (SFDQ)**, to further strengthen the DDoS protection provided by the puzzle framework. SFDQ is not only effective against DDoS attacks at multiple layers of the protocol stack, it is also simple to configure and deploy. SFDQ is implemented over a novel data structure, called **Indexed Linked List**, to provide enqueue, dequeue, and remove operations with $O(1)$ time complexity.

Keywords: Internet, availability, denial of service, distributed denial of service, replay attacks, cryptographic puzzles, tour puzzles, productive puzzles, stochastic fair drop, fair resource allocation, filtering, auto expire cache, indexed linked list.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

1.0 INTRODUCTION

As Internet is increasingly being used in almost every aspect of our lives, it is becoming a critical resource whose disruption has serious implications. Blocking availability of an Internet service may imply large financial losses, as in the case of an attack in 2000 that prevented users from having steady connectivity to major e-commerce Web sites, or it may imply threat to public safety as in the case of taking down of Houston port system in Texas [49] in 2003, or even to national security as in the case of Code Red worm attack against the White House Web site [43] in 2001.

Such attacks that aimed at blocking availability of computer systems or services are generally referred to as denial of service (DoS) attacks. As more and more essential services become reliant on the Internet as part of their communication infrastructure, the consequences of denial of service attacks can be very damaging. Therefore, it is crucial to deter, or otherwise minimize, the impact of denial of service attacks.

The original aim of the Internet was to provide an open and scalable network among research and educational communities [46]. In this environment, security issues were less of a concern. Unfortunately, with the rapid growth of the Internet over the past decade, the number of attacks on the Internet has also increased rapidly. CERT Coordination Center reported that the number of reported Internet security incidents has jumped from six in 1988 to 137,529 in 2003 [16]. The annual Computer Security Institute (CSI) computer crime and security survey reported that 30–40% of the survey participants were targeted by a DoS attack between 1999 and 2005 [28], and 21–29% of the participants were targeted by a DoS attack during 2006 to 2009 time period [57]. The 2010 Worldwide Infrastructure Security Report [22] found that DoS attacks had gone mainstream, and network operators were facing larger, more frequent DDoS attacks. The volume of the largest single attack observed in 2010

period reached a staggering 100 Gbps point, a 1000 percent increase since 2005 [22].

Preventing denial of service attacks can be very challenging, as they can take place even in the absence of vulnerabilities in a system. Meanwhile, it is extremely difficult, if not impossible, to precisely differentiate all attacker's requests from other benign requests. Thus, solutions that rely on detecting and filtering attacker's requests have limited effectiveness.

In the past, many DoS attacks have targeted the network bandwidth of Internet systems. However, with the increasing computational complexity of Internet applications and the increasing abundance of network bandwidth in the systems hosting these applications, server resources such as processing power, memory, or I/O bandwidth can become the bottleneck much before the network [62]. The most recent survey findings also confirmed that the application layer DDoS attacks are increasing in sophistication and operational impact [22].

The main purpose of this thesis is to investigate the problem of denial of service attack against public services in the Internet with a focus on mitigating application layer DoS attacks, and to design and evaluate a defense framework against DoS attacks.

1.1 AVAILABILITY AND DENIAL OF SERVICE

Availability is one of the three main objectives of computer security, along with confidentiality and integrity. Bishop [10] defines availability as the ability to use the information or resource desired. However, this description of availability lacks an important aspect of availability – timeliness. According to the Code of Laws of the United States (44 U.S.C § 3542 (b) (1)), "availability means ensuring timely and reliable access to and use of information." We believe this is a more accurate description of availability, hence, refine this description to define availability as the following.

Definition 1.1. Availability *is the ability to use the desired information or resource in a reliable and timely manner.*

Denial of Service is a threat that potentially violates the availability of a resource in a system. A *Denial of Service Attack*, on the other hand, is an action (or set of actions)

executed by a malicious entity to make a resource unavailable to its intended users. Gligor defines denial of service as follows [78]: "a group of otherwise-authorized users of a specified service is said to deny service to another group of otherwise-authorized users if the former group makes the specified service unavailable to the latter group for a period of time that exceeds the intended (and advertised) waiting time." This definition of denial of service takes into account the timeliness aspect of availability, and we refine it to define denial of service attack as follows.

Definition 1.2. Denial of Service Attack *is action(s) by a malicious entity to cause a targeted service to become unavailable to its users for an excessive period of time, typically, a time that significantly exceeds the intended waiting time.*

1.2 DOS ATTACK TYPES

There are three basic types of attacks [15] : 1) consumption of scarce, limited, or non-renewable resources, 2) destruction or alteration of configuration information, 3) physical destruction or alteration of network components. In this proposal, we focus on addressing the first type of attacks, i.e. attacks that consume a scarce, limited or non-renewable resource. The targeted resource that concerns us the most are processing power and memory.

In addition to the three basic types, DoS attacks can be classified into various categories based on different criteria. In [2], we proposed a taxonomy of DoS attacks and discussed each category of attacks in detail using examples. Figure 1.1 illustrates this taxonomy.

The targeted victim of a DoS attack can be an end system, a router, an ongoing communication, a link or an entire network, an infrastructure, or any combination of or variant on these [31]. In the case of an end system, the targeted victim can be an operating system or application.

Due to the large number of DoS attack categories, here we briefly discuss only the categories that are most relevant to our study. Refer to our technical report [2] for a detailed discussion of the complete taxonomy.

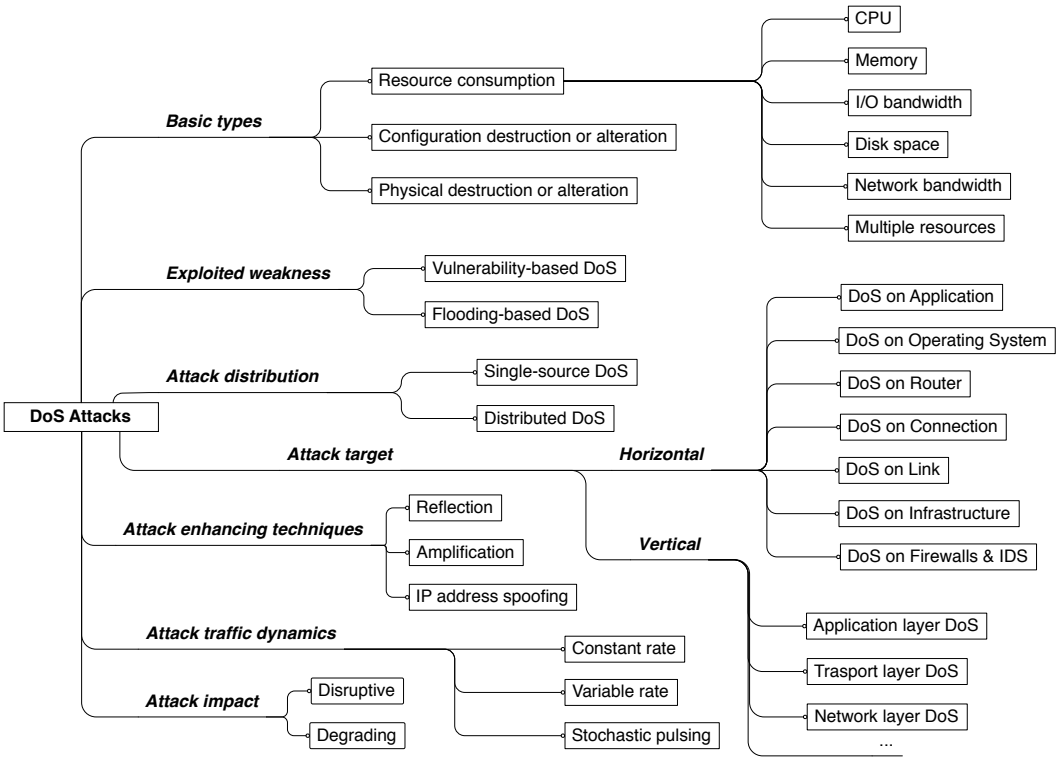


Figure 1.1: A practical taxonomy of denial of service attacks

1.2.1 Vulnerability and Flooding based Attacks

The different types of denial of service attacks can be broadly classified into *vulnerability based attacks* and *flooding based attacks*. A vulnerability based DoS attack exploits one or more flaws in a policy or in the mechanism that enforces the policy, or a bug in the software that implements the target system/service, and aims to excessively consume the resources of the target by sending it a few carefully crafted requests. For example, in an Exponential Entity Expansion attack, an attacker passes to an XML parser a small XML document that is both well-formed and valid, but expands to a very large file [68]. When the parser attempts to parse the XML, it ends up consuming all memory available to the parser application.

A DoS flooding attack, on the other hand, aims to deny service to legitimate users of a service by invoking vast amount of seemingly valid service requests and trying to exhaust a key resource of the target. For example, in a HTTP flooding attack, an attacker may send many requests to download large files from a Web server and saturate the server's upstream bandwidth, causing the server to deny service to benign requests.

Even in a scenario, where all software vulnerabilities and protocol flaws are eliminated, flooding based DoS attacks may still take place. However, for flooding attacks to be effective, the volume of the attack requests must be large enough to saturate the service capacity. Often times, it is hard for attackers to overwhelm the server by sending the traffic flood from a single client computer, since servers are generally better provisioned than clients. Even if an attacker controls a client computer that is as strongly provisioned as the target server, sending requests from a single client at a rate that saturates the full capacity of the server may easily be detected due to its exceptionally high rate. Therefore, effective flooding attacks are usually carried out by large number of computers that are compromised by attackers and are distributed in the Internet. The next classification of DoS attacks concerns the distribution of the attack source.

1.2.2 Single Source and Distributed Attacks

In a denial of service attack, attackers may launch their attacks from a single host or from multiple hosts that they control. When attacker's attack messages are originated from

multiple hosts that are distributed in the network, it is called a *distributed denial of service (DDoS) attack*. In contrast, when attacker's attack messages are generated by a single host, we call it a *single-source denial of service (SDoS) attack*.

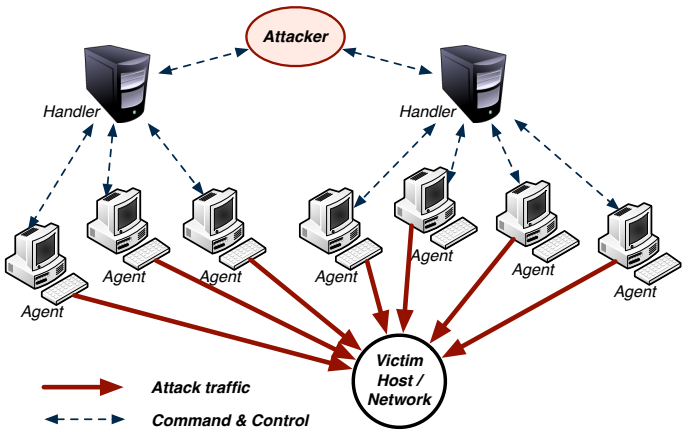


Figure 1.2: Distributed Denial of Service Attack

Generally speaking, DDoS attacks are more powerful than SDoS attacks, since the amount of processing power, memory, and bandwidth of a single computer hardly surpasses the combined resources of hundreds or thousands of compromised computers. In practice, defending against DDoS attacks is proven to be harder than defending against SDoS attacks. In this proposed work, we focus on mitigating the DDoS attack.

1.2.3 Application Layer DDoS Attacks

Application layer DDoS attacks, also referred to as service layer DDoS attacks, aim to make an application layer service unavailable to its intended users by exhausting one ore more key resources of the service. The commonly targeted resources are processing power (CPU), memory, I/O bandwidth, and network bandwidth.

HTTP flooding attacks [61], Session Initiation Protocol (SIP) flooding attacks [67], DNS flooding attacks [6] are some of the examples of application layer DoS attacks. Notable recent DDoS attacks that involve application layer DDoS include the DDoS attack against Wikileaks website in November 2010 [39] and following retaliatory DDoS attacks, know as

"Operation Payback", in December 2010 [51, 74].

In the past, many DoS attacks have targeted network bandwidth of Internet systems. However, with increasing computational complexity in Internet applications as well as larger network bandwidth in the systems hosting these applications, server resources such as CPU, memory, or I/O bandwidth can become the bottleneck before the network [62].

Especially for Web applications, the bottleneck in accessing a service is shifting away from network bandwidth to the computing resources available to the service. For example, analysis of Internet flooding events shows that peak line rates can exceed 600,000 packets per second [53], while even a highly replicated content-distribution service with 64 servers only reaches 40,000 requests per second with 6KB average request size [70].

Furthermore, much of the Web content is now generated dynamically using server side scripting or other application server methods. Most applications require interfacing with a database server. The increasing complexity, scale and size of offered Web services adds towards the higher resource consumption and hence bottlenecks at the end-system.

Most of the previous DDoS research efforts have focused on the network layer, and few research has been done on the mitigation of application layer DDoS attacks. One of the main reasons behind the conserved attention to the application layer DDoS is the infrequency of the application layer DDoS compared with network level flooding attacks. However, key findings of recent surveys that are based on more than 5,000 confirmed DDoS attacks suggested that application layer DDoS attacks are increasing in size, sophistication, and operational impact [22, 38].

Application layer DDoS attacks generally require far less bandwidth to be effective [38], making them a more cost effective DDoS strategy. Meanwhile, application layer DDoS attacks use legitimate TCP or UDP connections and can arbitrarily emulate the request syntax and network-level traffic characteristics of legitimate service requests, thereby making them much harder to detect.

1.3 PROBLEM SCOPE & DESGIN OBJECTIVES

1.3.1 Problem Scope

In this study, we focus on mitigating *application layer distributed service flooding attacks* that are aimed at public Internet services.

By *application layer*, we consider the layer 7 or above in the OSI Reference Model which provides services directly to the end users. For example, Web, email, file transfer services etc. Meanwhile, we do not aim to address specific vulnerabilities in specific applications or protocols. Rather, we aim to provide a more generic framework which can be customized to fit a specific application or service for the purpose of defending against DDoS. Since Web is one of the most popular Internet applications, it will be the main use case of our defense framework. However, that should not restrict the applicability of our framework to other Internet applications.

By *distributed*, we refer to the distributed DoS attack that is defined in the previous section.

By *service flooding*, we refer to the flooding based DoS attack that is also defined in the previous section. We do not consider vulnerability based attacks, because attack requests in such attacks often contain exploits that are targeted at a specific flaw in the target system (e.g., buffer overflows), and detecting such exploits is the goal of many intrusion detection and application firewall systems. Although it may be hard to detect the existence of certain exploits in attacker's requests, but the fact that the exploit exists in the attacker's requests differentiates them from the legitimate requests. In flooding based DDoS attacks, on the other hand, attacker's service requests differ from the legitimate requests in intent but not in characteristics, thus making such attacks more challenging to defend against.

Lastly, *public Internet services* refers to the Internet services and applications that are potentially accessible by all Internet users. In our view, a service that requires a user account to login is still a public service, if acquiring such an account is openly available to all potential users who are connected to the Internet. Services provided on private internets, private enterprise networks, and overlay networks etc., are not considered as public Internet

services.

In terms of the resources targeted by an attack, we focus on protecting the resources that are essential to processing a service request at the server. Processing a request often involves a combination of processor, memory, I/O bandwidth etc., and protecting the bottleneck resource should suffice for the type of DDoS attacks we are considering.

1.3.2 Design Objectives

Let us first take a look at a common puzzle-based DDoS defense and its problems in order to motivate our design goals.

In a computational puzzle protocol, a client is required to solve a moderately hard computational problem called *puzzle* and submit the solution as proof of work prior to its request being serviced by the server. For example, in a hash reversal puzzle (similar to the ones used in [26, 35, 71]), the server computes an m-bit Message Authentication Code (MAC) h using client's request message and timestamp as input, i.e. $h = MAC_K(request \mid\mid timestamp)$, and computes the hash digest h' of h by $h' = hash(h)$, where $hash$ is a cryptographic hash function such as SHA-1 [13]. The server then splits h into h_1 and h_2, whose lengths are r and $m - r$ bits respectively, and sends h_2, r, and the hash digest h' to the client as a puzzle (shown in Figure 1.3). The client then brute-force searchers for an r-bit long x, such that $hash(x \mid\mid h_2)$ is equal to h', and sends the satisfying x together with h_2 as the puzzle solution to the server. After receiving the puzzle solution submitted by the client, the server can re-compute h' without memorizing it , and checks to see if $hash(x \mid\mid h_2)$ is equal to h', and grants service to the client if that is the case. Figure 1.3 illustrates this computational puzzle protocol.

In the above hash reversal puzzle protocol, server can increase/decrease the amount of computation the client has to perform by increasing/decreasing the bit-length (r) of the missing part (h_1) of the puzzle. This length of the missing part is usually referred to as the *puzzle difficulty*.

Computational puzzle protocols can mitigate the effect of DDoS, because the more an attacker wants to overwhelm the server, the more puzzles it has to compute, hence more of

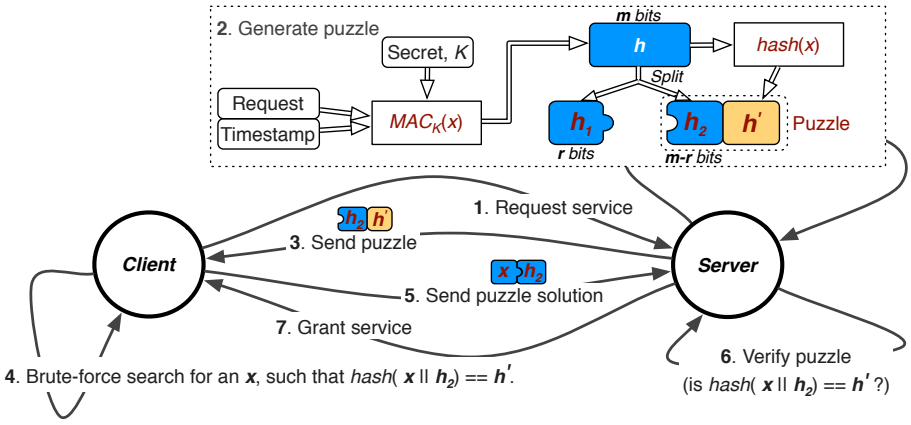


Figure 1.3: Overview of a common hash reversal puzzle protocol

its own computational resources must be expended. However, due to the variation in the computational powers of clients, the clients with powerful computational resources can solve puzzles at much higher rate than the destitute clients, thus getting the server to process their requests much more often than the requests of weaker clients.

Another crucial shortcoming of computational puzzle protocols is that all clients, including all legitimate clients, are required to perform such CPU-intensive computations that provide no utility to any of the parties involved. In the above hash-reversal puzzle, for example, the puzzle solution obtained via performing a large number of hash computations is simply thrown away after its validity is confirmed by the server.

Studying the shortcomings of existing puzzles-based defense solutions, such as the computational puzzle protocol, provided us with important clues regarding what requirements a puzzle-driven defense framework must satisfy to be effective against application layer DDoS attacks. Next, we present four key design objectives of our puzzle-driven defense framework.

No reliance on detection The implied premise of most current detection schemes is that the characteristics of DDoS attack traffic differ from normal traffic [76], which may not hold since sophisticated application layer DDoS attacks can mimic the service request behavior of legitimate clients at will. Therefore, our defense framework should be able

to mitigate the effects of DDoS attacks without relying on detection mechanism.

Minimize useless work Existing puzzle-based DDoS defense solutions imposes on clients CPU-intensive or memory-intensive computations, that are wasteful and do not provide utility to any involved parties of the puzzle protocol. Our puzzle-driven framework must minimize such wasteful computations. Meanwhile, we consider how to replace wasteful puzzle computations with computation tasks that are sub-processes of a useful application or service.

Fairness to all clients The puzzle-driven defense framework must be agnostic to the variation or asymmetry in the computational resources of client computers, in order to be effective against DDoS attacks in practical deployments. In other words, the framework must not discriminate against clients with limited computational power, and must guarantee each client a fair share of the server's capacity. We do not aim to achieve perfect fairness, instead the puzzle scheme must be designed such that a puzzle should take approximately the same amount of time to compute by any client, regardless of the CPU, memory, and bandwidth available to that client.

Minimum modification to client For a DDoS solution to be successful, it must be easily adaptable and deployable in practice. Easy adaptability necessitates minimum modification to both client and server programs. Especially, a client must be able to interact with the server without requiring functional changes. Although we plan on evaluating the framework in an experimental environment, we require the puzzle-driven framework to be designed such that it provides transparency to the client software.

1.4 KEY CONTRIBUTIONS

We start out by analyzing the strengths and weaknesses of existing DDoS defense mechanisms, and in particular, point out some of the key limitations of one of the promising DDoS solutions — puzzle based DDoS defense. We address these limitations and incorporate our ideas into a better puzzle defense solution that we call *puzzle+*. We show that puzzle+ framework achieves significantly better defense than existing puzzle based defense solutions.

Our second framework, called productive puzzle framework, improves upon the computation puzzle frameworks by replacing the useless computations with computations that contribute toward solving meaningful tasks. We give detailed mathematical analysis of the probability of cheating and error rate when using productive puzzles, and compare its effectiveness against DDoS attacks with that of puzzle+. We show that productive puzzles can still achieve very effectiveness defense against DDoS attacks even when the attackers upgrade their attacks with colluding and other cheating mechanisms.

Our third framework proposes another novel idea of delay-based puzzles that replaces expensive CPU-bound and memory bound puzzle computations at the client. When the number of malicious clients that can do two-way address spoofing is limited, this framework can effectively thwart the attack and guarantee each client an approximately fair share of the server's capacity, regardless of the disparity in the computational powers of clients.

Our last framework, also the most effective one, tightly integrates an efficient and fair resource allocation using a novel queuing mechanisms called *Stochastic Fair Drop Queue (SFDQ)* with a very effective blacklisting strategy that works hand-in-hand with SFDQ. This framework is not affected by the computational power of malicious attackers, nor does it susceptible to address spoofing. Although the idea was proposed while addressing the application layer DDoS attacks, this framework works equally well in the network layer and in other circumstances that involve queues.

1.5 ROADMAP OF THE DISSERTATION

Rest of this thesis is organized as follows. Chapter 2 surveys various existing DoS defense solutions, with a focus on puzzle based DDoS defense as well as fair resource allocation in the face of misbehaving or malicious entities. In Chapter 3, we introduce a client-server based model puzzle based DDoS defense system. Furthermore, we describe a threat model as well as the experimental framework we use to evaluate various DDoS defense solutions. In Chapter 4, we identify problems that are limiting the effectiveness of computation based puzzle schemes, and introduce *Puzzle+* scheme to address these limitations. We also study

the resource disparity and wasted computation problems with the computation based puzzle mechanisms. To address such problems, we introduce *Guided Tour Puzzles* in Chapter 6 and *Productive Puzzles* in Chapter 5 and show how they will address resource disparity or wasted computation problems. We break away from puzzle based DDoS defense in Chapter 7, and focus on how to combine queue management techniques with simple blacklisting mechanisms to achieve a more robust and more widely applicable DDoS defense solution. We summarize our work and conclude our thesis in Chapter 8.

2.0 SURVEY OF RELATED WORK

A plethora of solutions have been proposed in research literature to tackle the denial of service problem. In this chapter, we look at various existing solutions for defending against denial of service attacks, summarize techniques used in these solutions, and evaluate their strength and weaknesses. Due to the excessive number of existing proposals, we focus our attention to some of the representative work in each category of defense strategy. As some of the presented solutions can be argued to belong to multiple categories, we do not claim that the categorization given here provides a precise taxonomy. Rather, it serves as a way of effectively organizing voluminous related work.

2.1 DOS DEFENSE CHALLENGES

Although the original design goals of the Internet recognized the need to be robust in the presence of external attack, there was no equivalent concern with regard to the possibility of attacks by the Internets own users [46]. Under this "benign user" worldview, provisions to track and prevent malicious user behavior were never designed or implemented. Some of the challenges in defending against DDoS stems from the lack of security in the original Internet architecture, others are inherent to the general DoS problem. These challenges need to be well understood in order to design solutions that fundamentally address the problem, as well as to guarantee practicality of the solutions.

Here, we give a brief overview of such challenges. For a detailed discussion of the complete challenges, refer to our technical report [2].

Difficulty of distinguishing malicious requests It is difficult to distinguish between malicious requests and legitimate ones. This is true for packets, network flows, transport layer segments, or application service request messages. Even if certain malicious behavior can be reliably detected by signature based attack detection mechanisms, attackers can modify the characteristics of their attack messages to evade the detection. Although anomaly based detection mechanisms can detect unknown attacks, they are not very reliable due to the possibility of misidentifying normal behavior as an attack. The detection becomes enormously difficult when it comes to highly distributed flooding attacks, since such attacks do not have to restrict their attack messages to exploit certain vulnerability. Such flooding attack packets need not be malformed (e.g., contain invalid fragmentation field or a malicious packet payload) to be effective. In other words, attackers are free to create attack message that are indiscernible from legitimate request messages. Lastly, in principle it is not possible to distinguish between a sufficiently subtle DoS attack and a flash crowd [31].

Asymmetry in request and response overhead Asymmetry of request and response overhead refers to the asymmetry in the amount of consumed resources for generating a request at the client and creating its corresponding response at the server. In most cases, a client spends trivial amount of CPU and memory resources to generate a request, and the operations carried out by the server to produce the corresponding response incurs significantly more resource overhead in comparison. Making matters worse, attackers can create their malicious requests off-line prior to the attack, further minimizing the overhead of generating a service request.

Decentralized management An important design goal of the Internet architecture is that it must permit distributed management of its resources [17]. Current Internet can be seen as interconnection of many Autonomous Systems (AS), where each autonomous system is a set of routers and links under a single technical administration. Each autonomous system defines its own set of operating policy and security policy. The enforcement of a global security policy or mechanisms is enormously difficult, which makes solutions that require cross-domain cooperation unattractive. On the other hand, many distributed denial of service attacks may not be mitigated at a single-point, and require the defense

mechanisms to be deployed at multiple locations in the Internet. Designing solutions that can satisfy these conflicting requirements is hard.

Source address spoofing Users with sufficient privileges on a host can generate IP packets with source IP address field set to an address other than the legally-assigned address of that host. This is called *IP address spoofing*. IP address spoofing is frequently used in denial of service attacks. Attackers use IP address spoofing to hide the true origin of attack messages, or they can amplify or reflect attack traffic using address spoofing. Attackers can use multiple spoofed source addresses for the attack traffic originating from the same attacking machine to achieve diffusion of traffic floods, making threshold based rate-limiting and attack detection mechanisms ineffective.

Difficulties in DoS defense research The advance of DoS defense research has been hindered by the lack of attack information, the absence of standardized evaluation, and the difficulty of large-scale testing [52]. Very limited information about DoS incidents are publicly available due to organizations' unwillingness to disclose the occurrence of an attack, for fear of damaging the business reputation of the victim. Without detailed analysis of real-world DoS attacks, it is difficult to design imaginative solutions to the problem. In terms of standardized evaluation, there is no standard for evaluating the effectiveness of a DoS defense system. This makes it very difficult to compare the performance of various solutions. Moreover, the testing of DoS solutions in a realistic environment is immensely challenging, due to the lack of large-scale test beds or detailed and realistic simulation tools that can support Internet-scale network of nodes.

2.2 DEFENSE STRATEGIES

The strategies of various denial of service defense mechanisms can be broadly divided into four categories: *prevention*, *detection*, *response*, and *tolerance*. *Prevention* approaches attempt to eliminate the possibility of DoS attacks or prevent the attack from causing any significant damage. *Detection* can be further classified as *attack detection* and *attack source identification*. *Attack detection* monitors and analyzes events in a system to discover mali-

cious attempts to cause denial of service. It is an important step before directing further actions to counter an attack. *Attack source identification*, on the other hand, aims to locate the attack sources even when the source address field of malicious requests contain fake or erroneous information. *Response* mechanisms are usually initiated after the detection of an attack to eliminate or minimize the impact of the attack on the victim. *Tolerance* aims to minimize the damage caused by a DoS attack without being able to differentiate malicious actions from legitimate ones. It may suffice to merely know that system load is above certain threshold, in order to initiate the tolerance mechanisms.

For each of the four broad defense categories, we can further divide them into different defense mechanism types, based on the similarity of different solutions. Figure 2.1 illustrates the taxonomy of defense mechanisms that we created to classify the existing DoS solutions.

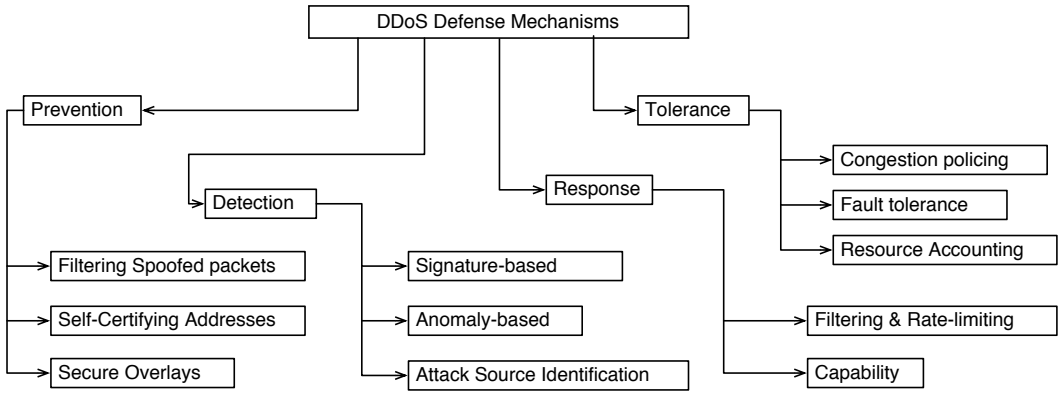


Figure 2.1: A taxonomy of DDoS Defense mechanisms

In this related work, we only discuss the "resource accounting" sub-category of the "tolerance" mechanisms category. For a complete discussion of all types of defense mechanisms illustrated in Figure 2.1, see our technical report [2].

2.3 TOLERANCE MECHANISMS

Tolerance mechanisms aim to minimize the damage caused by a DoS attack without being able to differentiate malicious behavior from legitimate ones. The obvious advantage of tolerance mechanisms is that they do not rely on detection mechanisms to identify attack, and in some cases they do not even need to know that an attack is happening. This is very helpful where detection of an attack and separating attack traffic or malicious service requests is especially hard, or when the accuracy of detection is low. Tolerance mechanisms recognize the unattainability of complete DoS prevention or detection, and focuses on minimizing the attack impact and maximizing the quality of service provided during the attack.

Existing approaches to DoS attack tolerance can be roughly summarized into several categories. They are congestion policing, fault tolerance, and resource accounting. In this section, we look at puzzle based DoS tolerance mechanisms.

2.3.1 Resource Accounting: Puzzles

Cryptographic puzzle approaches add resiliency to the protected system, as they try to minimize the effects of an attack on legitimate users of a system without being able to identify malicious clients from legitimate ones.

Since being introduced by Dwork and Naor to combat junk e-mails [24], cryptographic puzzles have been extended to defeat various attacks such as denial of service [35] [26] [71] [72] [73], Sybil attacks [12] [64], etc. Furthermore, many new ways of constructing and distributing puzzles have been introduced [71] [1] [23] [47] [29].

2.3.1.1 Client Puzzles Dwork and Noar [24] were the first to introduce the concept of requiring a client to compute a moderately hard but not intractable function, in order to gain access to a shared resource. However this scheme is not suitable for defending against the common form of DoS attack due to its vulnerability to puzzle solution pre-computations.

Juels and Brainard [35] introduced a hash function based puzzle scheme, called *client puzzles*, to defend against connection depletion attack. Client puzzles addresses the problem

of puzzle pre-computation. Aura et al. [4] extended the client puzzles to defend DoS attacks against authentication protocols, and Dean and Stubblefield [19] implemented a DoS resistant TLS protocol with the client puzzle extension. Wang and Reiter [71] further extended the client puzzles to prevention of TCP SYN flooding, by introducing the concept of *puzzle auction*. Price [60] explored a weakness of the client puzzles and its above mentioned extensions, and provided a fix for the problem by including contribution from the client during puzzle generation.

Waters et al. [73] proposed outsourcing of puzzle distribution to an external service called *bastion*, in order to secure puzzle distribution from DoS attacks. However, the central puzzle distribution can be the single point of failure, and the outsourcing scheme is also vulnerable to the attack introduced by Price [60].

Wang and Reiter [72] used a hash-based puzzle scheme to prevent bandwidth-exhaustion attacks at the network layer. Feng et al. [25] argued that a puzzle scheme should be placed at the network layer in order to prevent attacks against a wide range of applications and protocols. And Feng and Kaiser et al. [26] implemented a hint-based hash reversal puzzle at the IP layer to prevent attackers from thwarting application or transport layer puzzle defense mechanisms.

Portcullis [55] by Parno et al. used a puzzle scheme similar to the puzzle auction by Wang [71] to prevent denial-of-capability attacks that prevent clients from setting up capabilities to send prioritized packets in the network. In Portcullis, clients that are willing to solve harder puzzles that require more computation are given higher priority, thus potentially giving unfair advantage to powerful attackers.

In all of proposals above, finding the puzzle solution is parallelizable. Thus an attacker can obtain the puzzle solution faster by computing it in parallel using multiple machines. Morever, they all suffer from the resource disparity problem, and interferes with the concurrently running user applications. In comparison, guided tour puzzles are non-parallelizable, and addresses the problems of resource disparity and interference with user applications.

2.3.1.2 Non-Parallelizable Puzzles

Non-parallelizable puzzles prevents a DDoS attacker that uses parallel computing with large number of compromised clients to solve puz-

zles significantly faster than average clients. Rivest at al. [63] designed a *time-lock puzzle* which achieved non-parallelizability due to the lack of known method of parallelizing repeated modular squaring to a large degree [63]. However, time-lock puzzles are not very suitable for DoS defense because of the high cost of puzzle generation and verification at the server.

Ma [47] proposed using *hash-chain-reversal puzzles* in the network layer to prevent against DDoS attacks. Hash-chain-reversal puzzles have the property of non-parallelizability, because inverting the digest i in the chain cannot be started until the inversion of the digest $i + 1$ is completed. However, construction and verification of puzzle solution at the server is expensive. Furthermore, using a hash function with shorter digest length does not guarantee the intended computational effort at the client, whereas using a longer hash length makes the puzzle impossible to be solved within a reasonable time.

Another hash chain puzzle is proposed by Groza and Petrica [29]. Although this hash-chain puzzle provides non-parallelizability, it has several drawbacks. The puzzle construction and verification at the server is relatively expensive, and the transmission of a puzzle to client requires high-bandwidth consumption.

More recently Tritilanunt et al. [69] proposed a puzzle construction based on the subset sum problem, and suggested using an improved version [18] of *LLL lattice reduction* algorithm by Lenstra et al. [44] to compute the solution. However, the subset sum puzzles has problems such as high memory requirements and the failure of LLL in dealing with large instance and high density problems.

Although the non-parallelizable puzzles addresses one of the weaknesses of client puzzles discussed in Section 2.3.1.1, they still suffer from the resource disparity problem and interferes with the concurrently running user applications on client machines. Guided tour puzzles, on the other hand, address these two weaknesses of non-parallelizable puzzles.

2.3.1.3 Memory-Bound Puzzles

Abadi et al. [1] argued that memory access speed is more uniform than the CPU speed across different computer systems, and suggested using memory-bound function in puzzles to improve the uniformity of puzzle cost across different systems. Dwork et al. [23] further investigated Abadi's proposal and provided an abstract

memory-bound function with an amortized lower bound on the number of memory accesses required for the puzzle solution. Although these results are promising, there are several issues need to be solved regarding memory-bound puzzles.

First, memory-bound puzzles assume a upper-bound on the attacker machine's cache size, which might not hold as technology improves. Increasing this upper-bound based on the maximum cache size available makes the memory-bound puzzles too expensive to compute by average clients. Secondly, deployment of proposed memory-bound puzzle schemes require fine-tuning of various parameters based on a system's cache and memory configurations. Furthermore, puzzle construction in both schemes is expensive, and bandwidth consumption per puzzle transmission is high. Last, but not least, clients without enough memory resources, such as PDAs and cell phones, cannot utilize both puzzle schemes, hence require another service that performs the puzzle computation on their behalf.

3.0 SYSTEM AND THREAT MODEL

In this chapter, we introduce a simple yet practical model of client-server system that adopts a puzzle based defense. Furthermore, we provide a threat model that will be assumed for all of our puzzle based defense frameworks in this thesis.

3.1 SYSTEM MODEL

3.1.1 System Overview

We consider an Internet-scale distributed system of clients and servers. A *server* is a process that provides content or services to a large number of clients. A *client* is a process that requests service from a server. The term client and server are also used to denote the machines that runs the server process and the client process respectively. Clients are further classified as *legitimate clients* that do not contain any malicious logic and *malicious clients* that contain malicious logic. An attacker is a malicious entity who controls the malicious clients. In the denial of service context, a malicious client is commanded by the attacker to overwhelm the server with spurious request in order to deny or disrupt normal service for legitimate clients.

We use the term *transaction* to refer to a sequence of message exchanges between the client and the server that results in fulfillment or rejection of a single service request, as shown in Figure 3.1. The server may require the client to solve a puzzle which may involve interacting with the server or other proxies of the server multiple times, but the entire puzzle process is still considered as part of a single transaction. A single transaction in our system

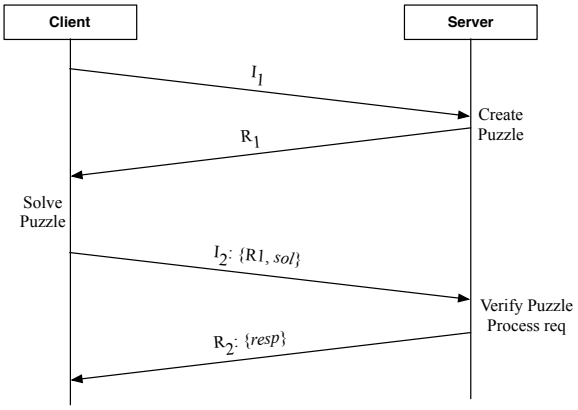


Figure 3.1: A single client-server transaction in a typical puzzle protocol.

usually has four messages: initial client request (I_1), initial server response (R_1) which may include a puzzle solving request, reinforced client request (I_2) that contains a puzzle solution, and final server response (R_2). When a puzzle requirement is not in effect, the transaction only includes I_1 and R_2. When the puzzle protocol involves client to interact with one or multiple proxies of the server, the set of messages included in a transaction takes the form of I_1, R_1, IP_1, RP_1, ..., IP_k, RP_k, I_2, R_2, where IP_1, RP_1, ..., IP_k, RP_k are messages exchanged between the client and the proxy of the server.

3.1.2 Mathematical Model

The mathematical model of characteristics of the system and the entities are described next.

3.1.2.1 Legitimate Clients To simplify our analysis, we assume that each legitimate client machine can process f_g (g as in *good*) instructions per second (i/s), and must perform c instructions for completing each transaction (i/tr), thus each client can achieve up to $\frac{f_g}{c}$ transactions per second (tr/s). This does not mean each client will certainly be sending requests at this high rate, and the actual rate can be much lower. When a computation puzzle scheme is in play, a client is required to solve a puzzle that takes τ instructions on

average, thus the average transaction rate of a client is given by

$$\frac{f_g}{c+\tau}. \tag{3.1}$$

Here, τ is also referred to as *puzzle difficulty* or *puzzle hardness*.

When a delay based puzzle scheme is in play, each puzzle takes δ seconds on average, thus each transaction takes $\delta + \frac{c}{f_g}$ seconds and the transaction rate becomes

$$\frac{1}{\delta + \frac{c}{f_g}} \tag{3.2}$$

$$=\frac{f_g}{\delta f_g + c}. \tag{3.3}$$

Assuming, the number of active legitimate clients in the system is N_g, the maximum load that can be generated by all legitimate clients in computation puzzle case is

$$L_g = N_g \frac{f_g}{c+\tau}, \tag{3.4}$$

and for delay based puzzles, it is

$$\hat{L_g} = N_g \frac{f_g}{\delta f_g + c}. \tag{3.5}$$

3.1.2.2 Malicious Clients Malicious clients can be modeled similarly, with the difference being the client machine processing power and the number of malicious clients. Assuming, the number of active malicious clients in the system is N_b (b as in *bad*) and each malicious client can process f_b instructions per second, the maximum load that can be generated by all malicious clients in computation puzzle case is

$$L_b = N_b \frac{f_b}{c+\tau}, \tag{3.6}$$

and for delay based puzzles, it is

$$\hat{L_b} = N_b \frac{f_b}{\delta f_b + c}. \tag{3.7}$$

3.1.2.3 Server The server can process F instructions per second, and each request takes C instructions on average to process by the server, thus the server has a capacity of $\mu = \frac{F}{C}$ transactions per second. According to the operating characteristics of a single server queue, the utilization factor of the server $\rho = \frac{\lambda}{\mu}$, where λ is the request arrival rates or offered load. We would like to utilize the puzzle scheme to control the utilization of the server, and to do so we must control λ. Here, the request arrival rate that we can control only refers to the arrival rate of service-eligible requests that accompanied by valid puzzle solution, as one cannot control how fast clients can send invalid requests.

Assuming, ρ is the current utilization of the server and ρ^* is the target utilization that we want to achieve, they can be written as

$$\rho = \frac{\lambda}{\mu}, \tag{3.8}$$

and

$$\rho^* = \frac{\lambda^*}{\mu}. \tag{3.9}$$

By combining (3.8) and (3.9), we can get

$$\lambda^* = \frac{\lambda\, \rho^*}{\rho} \tag{3.10}$$

λ^* is what the arrival rate should be if we want the utilization to be ρ^*, and the sum of loads offered by the legitimate and malicious clients should be equal to λ^*. Therefore, λ^* can also be written as

$$\lambda^* = L_g + L_b = N_g \frac{f_g}{c+\tau} + N_b \frac{f_b}{c+\tau} \tag{3.11}$$

for the computational puzzle case, and can be written as

$$\lambda^* = \hat{L_g} + \hat{L_b} = N_g \frac{f_g}{\delta f_g + c} + N_b \frac{f_b}{\delta f_b + c} \tag{3.12}$$

for the delay puzzles case.

Since we do not know the average CPU frequencies of legitimate and malicious clients, we can simply use an overall average CPU frequency f to replace them. Thus, the equation (3.11) becomes

$$\lambda^* = N_g \frac{f}{c+\tau} + N_b \frac{f}{c+\tau} = (N_g + N_b) \frac{f}{c+\tau} \tag{3.13}$$

We can replace the sum $(N_g + N_b)$ with N, which is the total number of clients currently active in the system, and set $c = 0$ since $c << \tau$. Subsequently, the equation (3.13) becomes

$$\lambda^* = N\frac{f}{\tau} \tag{3.14}$$

Solving the equation (3.14) for the average puzzle difficulty τ, we get

$$\tau = \frac{Nf}{\lambda^*} \tag{3.15}$$

As one may expect, the average puzzle difficulty should be collectively determined by the total number of active clients, the average processing power of the clients, and the target request arrival rate. Since, N is not a constant and changes of time, we can rewrite equation (3.15) as following to reflect the dynamicity of N:

$$\tau(t) = \frac{N(t)f}{\lambda^*}. \tag{3.16}$$

The puzzle difficulty formula (3.16) tells us how much puzzle computation per transaction each client should perform on average, but it doesn't tell us what the puzzle difficulty should be for each individual client. To determine the puzzle difficulty for each individual client, one must take into account the contribution of each client to the total offered load on the server.

Similarly, the puzzle difficulty δ in delay based puzzles can be derived from Equation (3.12) as following:

$$\delta(t) = \frac{N(t)}{\lambda^*}. \tag{3.17}$$

3.2 THREAT MODEL

It is assumed that network resources are large enough to handle all traffic, and the resource under attack is server computation. In particular, we assume that the malicious clients are Internet bots or zombie machines that are compromised and controlled by the attacker. The attacker can eavesdrop on all messages sent between a server and any legitimate client. We assume that the attacker can modify only a limited number of client messages that are sent to the server. This assumption is reasonable since if an attacker can modify all client messages, then it can trivially launch a DoS attack by dropping all messages sent by other clients.

To carry out an attack, the attacker can use one of, or a combination of, the following attack strategies:

- **Counterfeiting:** An attacker may send invalid puzzle solutions.
- **Time Shifting:** The attacker may collect large number of puzzle solutions before the attack, and use them to send large number of eligible requests during the attack.
- **Collusion:** The attacker may share puzzle solutions across malicious clients to reduce per-request computation overhead.
- **Replay:** The attacker may send the same valid puzzle solution multiple times.
- **Spoofing:** The attacker may use spoofed source addresses during an attack. This includes both *one-way spoofing* that fails the *Return Routability Test (RRP)* and limited *two-way spoofing* that can pass the test.
- **Concurrent Puzzle Solving:** The attacker may solve multiple puzzles concurrently if doing so can reduce the total amount of time to solve them all.

Attacks that mainly use one of these strategies can be referred to as *counterfeiting attack*, *time shifting attack*, *collusion attack* (or *cookie jar attack*), *replay attack*, *spoofing attack*, and *concurrent puzzle solving attack*, respectively. An attacker may also launch attacks on the puzzle scheme itself, including puzzle construction, puzzle distribution, or puzzle verification. We collectively refer to the attacks on the puzzle scheme as *puzzle resisting attacks*. We use several cryptographic algorithms in our schemes, but do not consider various brute-force or cryptanalysis attacks against the algorithms themselves. However, we do require the

construction, distribution and verification algorithms in our puzzle schemes to be efficient such that they do not become vulnerable to *algorithmic complexity attacks*.

3.3 EVALUATION FRAMEWORK AND METRICS

Most of our evaluations are carried out using the Network Simulator 2 (NS-2) [48] simulation environment. So, we describe the common simulation setup and evaluation metrics we used in this section to be shared by the later chapters, rather than repeating the description in each chapter where we have a simulation study. Specific simulation setup changes and changes in the evaluation metrics and the assumption we make in each chapter will be described in detail in the evaluation section of the corresponding chapter.

3.3.1 Experiment Methodology

We conduct experimental evaluation of the productive puzzle defense in a realistic simulation model we built in NS-2. To create a network topology that can closely resembles large-scale wide area networks, such as the Internet, a topology with $5,000$ nodes is generated using the Internet Topology Generator 3.0 (Inet-3.0) [75]. The bandwidth and the link delay values are calculated based on the Inet-3.0 generated link distance values. We also use a 342 node topology that we programmatically generate for many of our experiments if it becomes impractical to conduct thousands of experiments needed using the large-scale topology.

Since client and server nodes are located in the edge in real networks, we use degree-one nodes — nodes that have a single link to the rest of the network graph — from the generated topology as client and server nodes. We randomly choose a degree-one node as the server node and the remaining degree-one nodes are used as client nodes. The percentage of malicious client nodes is varied from 0% to 90% with an increment of 10%.

It is known that the Internet's self-similar traffic can be produced by multiplexing ON/OFF sources that have fixed rates in the ON periods and heavy-tail distributed ON/OFF period lengths [42] [56]. As such, clients are setup to generate ON/OFF traffic with Pareto

distributed ON/OFF period lengths to mimic the Internet traffic. Malicious clients generate traffic at 10 times the rate of legitimate clients.

Since NS-2 does not provide a CPU model, we model the server's CPU as a link with a certain bandwidth and zero propagation delay. This link is created between the server node and a dummy node that is connected only to the server. When a client request arrives at the server, the server injects a packet with its size equals to the server response size into the link toward the dummy node. And when the packet is pinged back by the dummy node (which implies the completion of the server processing of the request), the server sends a response to the client. The capacity of this link is set according to the CPU processing capacity that is being simulated. A round-robin queue is used as the server's request queue to model the CPU's round-robin process scheduling.

3.3.2 Evaluation Metrics

We mainly use three evaluation metrics — average completion time of a single legitimate request, percentage of the server CPU allocated to legitimate requests, and percentage of denied legitimate requests. The average completion time is calculated by recording the time spent between sending of a request and the receiving of its response, which includes the time spent on solving puzzles, for all completed requests of all the legitimate clients and taking the average. The percentage of the server CPU allocated to legitimate requests is computed as the fraction of the time the server's CPU is processing the requests of legitimate clients. The percentage of denied legitimate requests is computed by dividing the total number of legitimate requests denied service by the total number of legitimate requests sent.

4.0 PUZZLE+: AN IMPROVED COMPUTATIONAL PUZZLE FRAMEWORK

Cryptographic puzzles have been proposed to defend against DoS attacks with the aim of balancing the computational load of the server relative to the computational load of the client [35] [26] [4] [19] [71] [3]. Solving a puzzle typically requires performing large number of cryptographic operations, such as hashing, modular multiplication, etc. Thus, how many requests a client can get server to fulfill is limited by the computational resources available to the client, and consequently limiting the client's ability to DoS attack the server.

Although, puzzle based DDoS mitigation methods are promising, most of the existing work focus on puzzle construction, verification, and security analysis of puzzle protocol and few discuss the practical aspects puzzle based defense. Puzzle based defense mechanisms appear to be not widely deployed in practice, partly because of those issues. We identify three such problems that existing literature on puzzles do not adequately address: (1) there is no clear formula for determining the hardness of puzzles dynamically that takes into account the cost of processing the client request, the number of active clients in the system, and the current offered load of the server; (2) it is assumed that the puzzle defense should be switched on when the server is under attack, but it is unclear how to determine when the server is under attack; (3) few existing literature on puzzle based DDoS defense mentions the puzzle solution replay attack, and none propose a working solution to to prevent it.

In this chapter, we provide effective working solutions to these problems. Furthermore, we integrate these solutions into an improved puzzle scheme, that we call *Puzzle+*, and show that it is far more effective than the existing cryptographic puzzle based DDoS defense solutions.

4.1 PER-REQUEST PUZZLE HARDNESS

The concept of *puzzle hardness* or *puzzle difficulty* is mentioned in almost all puzzle literature, however, none gives a usable mathematical model or formula for computing it. Dean and Stubblefield [19] suggest using an empirical value of 20 bits for the hash reversal puzzles that they adopted for protecting against SSL based DoS attacks. Wang et al. [71] and Parno et al. [55] proposed to use an auction style determination of puzzle difficulty, where the clients, not the server, determine the hardness of the puzzles they solve to increase their chance of getting service. Using such mechanisms, a client has to do many attempts to before finding a puzzle difficulty that can allow them service; some subset of the client population may not be able to acquire service in cases where computationally strong attackers can raise the puzzle difficulty to levels that these clients cannot solve. The BitCoin backbone protocol utilizes a hash based puzzle mechanism and concludes the importance of calibrating the difficulty of puzzle [27], but do not provide how to calibrate it except suggesting to take into account the number of players in the system.

Laurie and Clayton [41] shows that cryptographic puzzles or proof of work systems do not work if puzzle difficulty is set too high. Groza and Warinschi [30] points out the lack of rigorous treatment of on the parameters (such as puzzle difficulty) of the puzzle based DoS defense proposals, and emphasizes the fact that most of the proposals are based on common sense, e.g., when a server is under attack the hardness of the PoW is increased, or on empirical observations, e.g., the best protection is achieved when the difficulty is set to a certain threshold. They provide a bound on the maximum puzzle difficulty level, and proves that there is no DoS protection benefit of setting the puzzle difficulty above that threshold. However, they do no provide a concrete mathematical model or algorithm for determining the puzzle difficulty under different system conditions either.

In this section, we start with the simple mathematical model of puzzle difficulty that we introduced in Section 3.1.2, and strengthen it by considering the dynamically changing load of the server and the normalized cost of the client request. We do not claim the resulting model is the best one for determining the puzzle difficulty, but we show that it works very well to provide strong defense against DDoS attacks while guaranteeing the target server

load.

4.1.1 Computing Puzzle Hardness

Recall that we derived a puzzle hardness model $\tau(t) = \frac{N(t)f}{\lambda^*}$ in Equation (3.16) in Section 3.1.2 of Chapter 3. Computing the target arrival rate λ^* is straightforward, as it is equal to $\lambda^* = \rho^*\mu$. The server capacity μ is known, and the target utilization ρ^* can be set to a value desired by the server operator. Estimating the number of currently active clients $N(t)$ is given in the next section. The average client CPU frequency f can be estimated using empirical data available.

However, the puzzle hardness τ given by $\tau(t) = \frac{N(t)f}{\lambda^*}$ does not make distinctions between different requests that incurs different levels of load on the server. Specifically a request that costs the server 1 second to compute will be given the same difficulty level puzzle that is given in response to a request that takes the server 10 millisecond to service. With no consideration given to the cost of a request, attackers can more easily overwhelm the server by sending mostly expensive requests. To address that, we incorporate a normalized request cost to the basic puzzle difficulty model in Equation (3.16) and arrive at the following formula:

$$d_{req} = \frac{N(t)f}{\rho^*\mu}\frac{t_{req}}{t_{avg}}, \tag{4.1}$$

where, t_{req} is the average time it takes the server to service request req, t_{avg} is the average time it takes to service any request, and $\frac{t_{req}}{t_{avg}}$ is the normalized cost of servicing the request req. Note that t_{req} is the cost of serving a specific type of request, for example, the cost of serving a specific query or Web page in the Web server example. t_{req} can be computed by the server by keeping records of the time spent processing the request req for the recent M occurrences and taking the average; whereas t_{avg} can be computed by taking the Exponential Moving Average of all recent requests processed by the server.

With this new puzzle hardness model, a request that costs k times more than the average request cost t_{avg} will get a k times harder puzzle to solve. That will help eliminate the advantage of attackers that leverage expensive requests.

The improved puzzle hardness model in (4.1), however, still does not take into account the current load of the server. As the server load is heavy, we want to increase the puzzle

hardness, so that it takes the client longer to return with a puzzle solution accompanying its request, that will decrease the number of service eligible requests arriving at the server in per unit of time; when the server load is light, it can afford to service more requests, therefore we can decrease the puzzle difficulty.

We capture this concept of dynamically adjusting the puzzle hardness based on the server load using what we called *Adjustment Factor* (AF), and arrive at our final puzzle hardness model as follows.

$$d_{req} = \frac{N(t)f}{\rho^*\mu}\frac{t_{req}}{t_{avg}}AF(t), \tag{4.2}$$

The Adjustment Factor $AF(t)$ not only adjusts the puzzle difficulty by accounting for the server load change, but it also makes up for the error in some of the estimated values in the puzzle hardness formula. $AF(t)$ is computed as follows.

$$\begin{cases} AF(t_0) = 1 \\ AF(t_i) = AF(t_{i-1}) + \varepsilon(t_i) \end{cases} \tag{4.3}$$

Basically, one AF value is computed for each time interval t_i. When the server first turns on puzzle defense at time t_0, $AF(t_0)$ is set to 1. After that, AF at time t_i is the sum of AF at the previous interval $AF(t_{i-1})$ plus a change $\varepsilon(t_i)$. ε should be positive when the current server load ρ is larger than the target load ρ^*, so that this positive change will increase AF and subsequently increase puzzle hardness; similarly, ε should be negative when $\rho \leq \rho^*$, so that it will decrease AF and puzzle hardness subsequently. Meanwhile, the size of the change ε should reflect the difference between ρ and ρ^*. We design a $\varepsilon(t)$ function that satisfies all of these requirements as follows.

$$\begin{cases} \varepsilon(t_i) = 1 - e^{\frac{\rho(t_i)}{\rho*}(\rho(t_i)-\rho*)}, \text{if } \rho(t_i) > \rho* \\ \varepsilon(t_i) = 1 - e^{\rho*(\rho(t_i)-\rho*)}, \text{if } \rho(t_i) \leq \rho* \end{cases} \tag{4.4}$$

Keep in mind that this is not the only change function that can be used, it is just one that worked well in our simulation study. Figure 4.1 plots the change function $\varepsilon(t)$ when $\rho^* = 0.7$, and shows that the $\varepsilon(t)$ function behaves in line with its requirements.

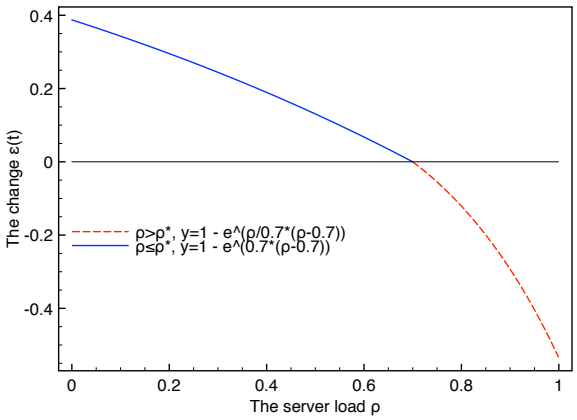


Figure 4.1: The change function $\varepsilon(t)$ when $\rho^* = 0.7$.

4.1.2 Estimating Number of Active Clients

The puzzle hardness model given by Equation (4.2) requires the number of clients $N(t)$ that are currently active in the system. A straightforward way to estimate this value is to choose a time internal length Δ seconds, and count the number of clients that received service in each interval; and use the Exponential Moving Average (EMA) technique to get a smoothed estimate of the number of active clients. Specifically the number of active clients value to be used in our puzzle hardness calculations, during the interval t_{i+1}, will be calculated at the end of the interval t_i as follows.

$$N_{avg}(t_i) = \alpha \times N(t_i) + (1 - \alpha) \times N_{avg}(t_{i-1}), \tag{4.5}$$

where, $N_{avg}(t_{i-1})$ is the estimated average number of active clients for the interval t_{i-1}.

The Exponential Moving Average can give us a pretty good estimate of the number of clients. However, it is not easy to pick a good value for the *smoothing factor* α. Moreover, the estimated value used in the current interval is computed in the last interval, as such it lags behind the true number of clients at the time of using the value in the puzzle hardness computation.

We found that we can have even better estimates if we are willing to incur some extra

memory cost. The idea is to use the *Auto-Expire Cache* that we are going to introduce in the next section. Auto-Expire Cache can keep the number of clients that were active in the last Δ seconds, and can automatically remove a client's ID from itself when that client did not come back during that last Δ period of time. As we show in the next section, Auto-Expire Cache can eliminate the expired items in constant time.

Estimating the number of clients using Auto-Expire Cache is easy: each time we service a client request, we add its ID to the Auto-Expire Cache, which will keep only one copy of each unique ID added to itself; when we need the number of active clients in the system, we simply call the GETNUMITEMS function of the Auto-Expire Cache. Since non-active clients' unique identifiers automatically get removed from the cache, the cache will always have the most recent set of active clients in the last Δ seconds.

The next section will give for more information on the Auto-Expire Cache.

4.2 PREVENTING PUZZLE SOLUTION REPLAY ATTACKS

In a puzzle protocol model, a malicious client may attempt to amplify the impact of its attack by repeatedly using the same puzzle solution. Specifically, the third message $I2$, as described in the puzzle protocol model in Section 3.1.1, that includes the puzzle solution can be replayed by a malicious client to the server until the corresponding puzzle is expired, as demonstrated in Figure 4.2.

Aura et al. [4] suggests checking for if a solution is replayed or not, but give no specific information on how that is enforced. Wang and Reiter [71] propose checking a newly admitted request's client nonce against all existing requests in the service queue to see if a request with that nonce already exist in the queue, and drops the request if exists. Such checks on queue data structure are expensive when the number of items in the queue is large, and replay is still possible if the attacker waits long enough between two replays.

Feng et al. [26] suggest binding the puzzle to specific flow or packet to require solving a new puzzle for each new flow or new packet. Lakshminarayanan et al. [40] suggested keeping each puzzle at the server until a reply is received from the client or a specified amount of

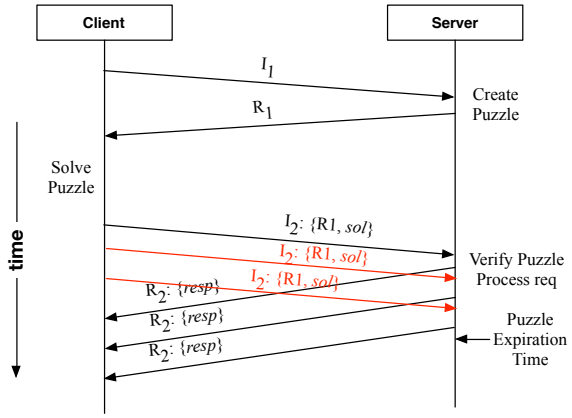


Figure 4.2: The puzzle solution replay attack in a puzzle protocol.

time has passed since its creation. All of these solutions require keeping the initial state at the server, however doing so can make the server susceptible to memory depletion type DoS attacks, similar to the TCP SYN flooding attacks [14].

4.2.1 Naive Solutions

In a typical cryptographic protocol setting, the replay attacks can be prevented by establishing a unique session ID for each run of the protocol and using client and server side nonces. However, keeping track of sessions and session IDs is susceptible to memory depletion attacks that target bookkeeping of the initial messages, since generating valid unique instances of the first message $I1$ in a puzzle protocol is trivial. In comparison, generating valid unique instances of $I2$ is a lot more expensive. So, instead of keeping the state for message $I1$, keeping unique instances of already used puzzle solution in message $I2$, one might be able to provide much better resistance against flooding attacks that target the memory resources. In fact keeping a unique identifier, such as a nonce, that corresponds to the puzzle solution should be enough.

The server, however, cannot keep these used unique identifiers forever, so one must combine keeping limited state with puzzle expiration time to achieve efficient and effective protection against replay attacks.

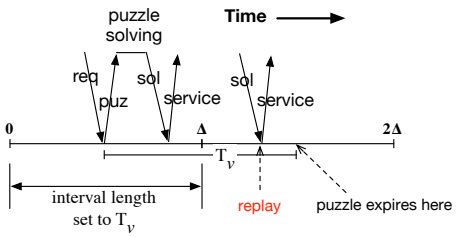


Figure 4.3: A replay attack against the wooden-man solution

A straw-man solution for keeping limited state is to use a hash table to keep the recently submitted puzzle solution, so that resubmitting an already used puzzle solution can be detected by looking it up in this hash table. 'Recently submitted puzzle solution' implies that there is a limit to how long a puzzle solution should be kept in the hash table. Ideally, a puzzle solution should be removed from the hash table right after the corresponding puzzle has expired, i.e., $now() > t_{exp}$, where t_{exp} denotes the puzzle expiration time. But doing so requires periodically scanning the hash table and removing expired entries, which is not trivial and requires $O(n)$ operations for each scan.

A wooden-man solution, an improvement over the straw-man solution, would be to use one fixed size hash table per interval, meaning a fixed size memory is allocated at the beginning of each interval and deallocated at the end of the interval. The length of the interval Δ should at least be equal to the length of the puzzle validity period T_v, note that the Δ here is independent of the Δ we used in estimating the number of active clients. As this solution quickly removes older entries at the end of each interval, memory requirement can be fairly small. However, this solution cannot prevent puzzle solution replay attacks entirely. Attackers can time their puzzle request towards the end of the current interval so that the current interval terminates soon after its request already being serviced once. The attacker can then replay the same solution in the next interval before the puzzle expires. Figure 4.3 illustrates this attack.

4.2.2 Auto-Expire Cache based Solution

We designed a data structure called *Auto-Expire Cache (AEC)* that can delete expired items in $O(1)$ time and uses fixed-size memory. The main idea of Auto-Expire Cache is to use two fixed size hash tables for each time period of length Δ, and release the memory occupied by the hash tables when their corresponding time period passes. During each time interval, the first hash table contains items added in the last and the current interval, whereas the second one contains items that are added in the current interval only. The operations in Auto-Expire Cache are described in Algorithms 4.1.

Algorithm 4.1: Auto-Expire Cache

```
Data: hashtable[0], hashtable[1] - the two underlying hash tables used by AEC;
      Δ - the cache entry expiration interval.
1  function AEC.addItem(item, current_time)
2      AEC.cleanExpiredTables(current_time)
3      add item to AEC.hashtable[0]
4      add item to AEC.hashtable[1]
5  function AEC.cleanExpiredTables(current_time)
6      if current_time ≥ AEC.lastClean + Δ then
7          delete AEC.hashtable[0]
8          AEC.hashtable[0] = AEC.hashtable[1]
9          AEC.hashtable[1] = newHashTable()
10         if current_time ≥ AEC.lastClean + 2Δ then
11             delete AEC.hashtable[0]
12             AEC.hashtable[0] = newHashTable()
13         numLaggedIntervals = floor((current_time - AEC.lastClean)/Δ)
14         AEC.lastClean = AEC.lastClean + numLaggedIntervals × Δ
15 function AEC.getNumItems(current_time)
16     AEC.cleanExpiredTables(current_time)
17     return AEC.hashtable[0].Size()
18 function AEC.Contains(item, current_time)
19     AEC.cleanExpiredTables(current_time)
20     return (AEC.hashtable[0].AEC.Contains(item) or AEC.hashtable[1].AEC.Contains(item))
```

The main operations in Auto-Expire Cache are addItem, getNumItems, and contains, all of which calls the cleanExpiredTables function at the beginning of the function. This is because Auto-Expire Cache adopts a lazy invalidation of cache entries, and checks to see if one or both of the hash tables expired before every main operation. In cleanExpiredTables function, if more than one time interval Δ has passed since the last cleaning of the cache, then the first hash table is deleted and pointed to the second hash table, and the second hash table is pointed to a new hash table; and if more than two time

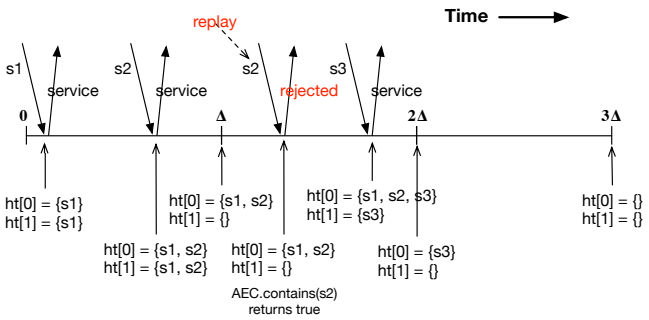


Figure 4.4: An example usage of Auto-Expire Cache

interval Δ has passed since the last cleaning, then both hash tables are deleted and are pointed to new hash tables. Here, *last cleaning* is the start of time interval that corresponds to the $hashtable[0]$. If Δ time or more has passed since the last cleaning, that means the time interval corresponding to the $hashtable[0]$ is in the past now, hence $hashtable[0]$ is expired and must be deleted. In the ADDITEM function, if the item is already exists in the hash table, it simply overwrites the same value and does not insert multiple copies of the same item. Note that if memory complexity of Auto-Expire Cache can further be minimized by using Bloom Filters [11] instead of the hash tables.

Auto-Expire Cache guarantees that an item inserted in it will stay in it for at least the length of one interval. And when we set the interval length equal to the puzzle validity period T_v, it will be guaranteed that an already used puzzle solution will stay in the cache at least after puzzle validity period expires. Auto-Expire Cache also guarantees that an item inserted in it will be removed from it after at most the length of two intervals. As such, expired entries are removed swiftly to save memory and prevent memory exhaustion attacks. An example usage of Auto-Expire Cache is given in Figure 4.4.

To prevent puzzle solution replay attacks, the server maintains an Auto-Expire Cache for keeping the already-used puzzle solution. Since each puzzle and its solution can be uniquely identified by a 128-bit *token*, saving this identifier in the cache, instead of the actual puzzle solution, would suffice for the purpose of preventing replay. During the puzzle protocol, the

server checks if the solution submitted by the client exists in the cache by calling CONTAINS function of Auto-Expire Cache, and saves the corresponding puzzle solution identifier in the cache if it doesn't exist; If the identifier does exist in the cache, the server rejects the client request.

As the puzzle expiration duration can vary for different puzzles with varying puzzle hardness, Auto-Expire Cache may prematurely expire a puzzle solution that has a long expiration time t_{exp} in some rare cases. Choosing a large value for the Auto-Expire Cache interval Δ could address the problem, but at the same time it increases the memory demand of the cache. An alternative is to use fixed puzzle validity period to avoid such cases.

4.3 DETERMINING PUZZLE SWITCH ON/OFF

Determining when to switch puzzles on or off correctly is important, because it allows puzzle based DoS defense to respond quickly when the protected server is under attack and eliminate the puzzle solving burden on clients when there is no attack.

Most existing puzzle literature either provides no information on how to determine the puzzle switch on/off times, or suggests to turn on the puzzle when the server is under attack or when the server's load is above certain threshold. Dean and Stubblefield [19] provide the most detailed discussion of this matter among the existing puzzle literature, and propose using two separate thresholds — one for turning puzzles on and one for turning them off. But they only provide specific threshold numbers that are suitable under very limited circumstances for the TLS [21] handshake they were trying to protect.

We propose a heuristic approach to determining puzzle switching point. Similar to Dean and Stubblefield's suggestion we use two threshold levels — puzzle turn on threshold λ_{on} and turn of threshold λ_{off}, but there are important differences. First, we compare the request arrival rates λ, i.e. the offered load of the server, to these two thresholds, instead of checking the server load or utilization factor ρ against the thresholds. In a DoS context, the offered load can get to a level that is multiple times of the server load before the server load reaches above our preset threshold, so it can indicate the presence of an attack sooner than the server

load does. Second, we set the puzzle turn-on threshold λ_{on} to the average expected load of the server, and set the turn-off threshold λ_{off} to the half of λ_{on}, i.e., $\lambda_{off} = \frac{1}{2}\lambda_{on}$.

The following is our algorithm for computing the server's offered load and determining the puzzle on/off:

Algorithm 4.2: Determining Puzzle Switch On/Off

```
1  function UPDATEOFFEREDLOAD()
2      λ_new = req_costs/load_interval/μ
3      λ = α × λ_new + (1 − α) × λ
4      HANDLELOADCHANGE()
5  function HANDLELOADCHANGE()
6      if λ > λ_on then
7          puzzle_on = true
8      if λ < 0.5 ∗ λ_on then
9          puzzle_on = false
```

4.4 PUZZLE+ FRAMEWORK

In this section, we first design introduce the design of a computation-based puzzle protocol, called *Puzzle+*. Puzzle+ integrates the puzzle hardness model, replay attack defense solution, and puzzle switch on/off algorithm that are introduced in previous sections. We adopt the commonly used *hash reversal* based puzzle construction mechanism, and build a puzzle protocol that is secure against the threat model that we defined in Section 3.2. Next, we evaluate the Puzzle+ protocol and compare its DDoS defense effectiveness against the existing computation-based puzzle protocols.

Similar to most puzzle protocols, the client-server interaction in Puzzle+ protocol consists of four message exchanges: (1) initial client request (I_1), (2) initial server response (R_1) which includes a puzzle, (3) reinforced client request (I_2) that contains a puzzle solution, and (4) final server response (R_2). Figure 4.5 illustrates the client server interaction in the Puzzle+ protocol.

The client's initial request message I_1 contains the following information:

$$I_1 = \{A_c, A_s, req, h_c\}, \tag{4.6}$$

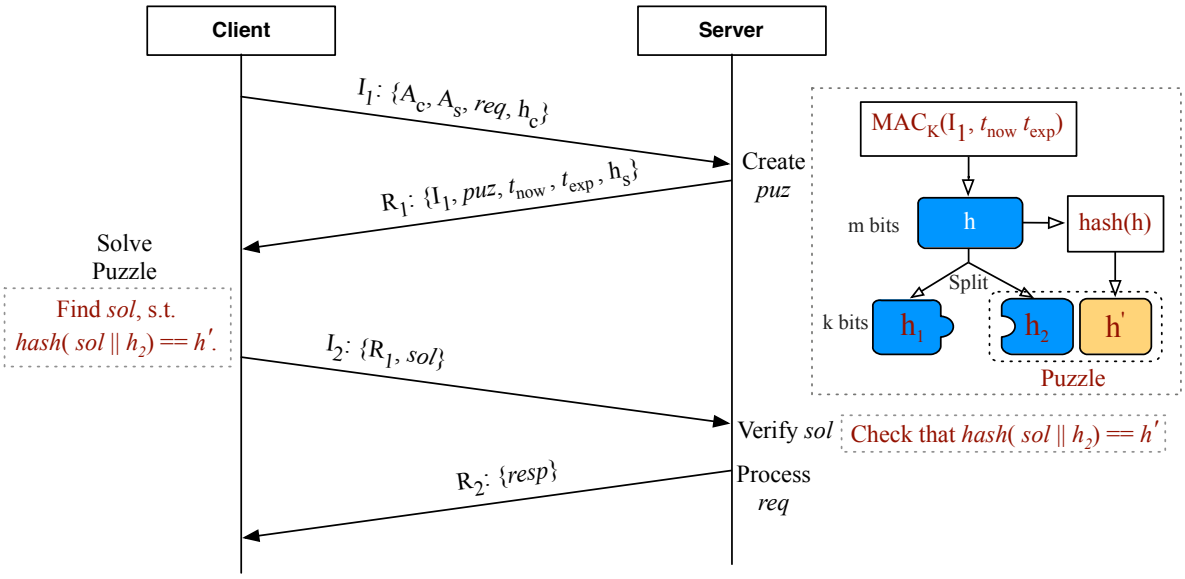


Figure 4.5: The client server interaction in Puzzle+ protocol

where A_c and A_s are unique identifiers of the client and the server, such as IP address or Uniform Resource Identifier (URI) [9]; req is the client's service request, for example a URI of a Web resource; h_c is an Hash-based Message Authentication Code (HMAC) [36] of the previous 3 items, and it is computed as following:

$$h_c = hmac(K_c,\ (A_c||A_s||req)) \tag{4.7}$$

where, K_c is HMAC secret key of the client, and $||$ is the string concatenation operator. h_c keeps the integrity of the client's message and also plays the role of a *client nonce*.

Upon receiving the client request, the server constructs a hash reversal puzzle puz as follows, if the puzzle mechanisms in turned on (the server uses Algorithms 4.2 introduced in Section 4.3 for determining puzzle on/off.)

$$puz = \{h_2, h'\}, \tag{4.8}$$

where, $h_2 = [h]_{m-k}$ represents the lower $m-k$ bits of h, $h' = hash(h)$, and h is an m-bit hash digest that is computed as following:

$$h = hmac(K_{puz}, (I_1, t_{now}, t_{exp})), \tag{4.9}$$

where, K_{puz} is the server's secret key that it only uses for generating puzzles, t_{now} is the time of the puzzle generation, and t_{exp} is the puzzle expiration time. The computation of puzzle puz is illustrated in Figure 4.5.

In essence, the hash reversal puzzle contains the output of hashing h and $m - k$ bits of h, and the client is expected to find the hidden k bits of h by doing an average of 2^{k-1} hash operations. Of course, the assumption here is that the client does not have a faster way to find the missing k bits, except for using brute-force search. The length of k determines how many hash operations the client needs to perform, hence determines the hardness of the puzzle. The puzzle hardness model we gave in Equation (4.2) computes the hardness in number of instructions, so it needs to be solved for k. Assuming computing a single hash requires C_{hash} instructions, then computing a hash reversal puzzle with k hidden bits takes $C_{hash} \cdot 2^{k-1}$ instructions, which is equal to d_{req} in Equation (4.2), i.e.,

$$C_{hash} \cdot 2^{k-1} = d_{req} \tag{4.10}$$

Solving this equation for k, we get

$$k = \log_2 \left(\frac{d_{req}}{C_{hash}} \right) + 1 \tag{4.11}$$

$$= \log_2 \left(\frac{\frac{N(t)f}{\rho^* \mu} \frac{t_{req}}{t_{avg}} AF(t)}{C_{hash}} \right) + 1 \tag{4.12}$$

Once the server constructed the puzzle puz, it replies with message $R1$, which contains the following information:

$$R_1 = \{I_1, puz, t_{exp}, uid_{puz}, h_s\}, \tag{4.13}$$

where puz is the hash reversal puzzle generated earlier, uid_{puz} is a 128-bit random string that is used as a unique identifier of the puzzle, h_s is the message authentication code of the entire message and also plays the role of a *server nonce*. h_s is computed as following:

$$h_s = hmac(K_s, \ (I_1 || puz || t_{exp} || uid_{puz})), \tag{4.14}$$

where K_s is the HMAC secret key of the server.

After receiving the puzzle, the client computes puzzle solution *sol* by brute force searching for a value that satisfies $hash(sol||h_2) == h'$. Once, the client finds a *sol* that satisfies the equality, it sends that solution value along with its original service request to the server in message I_2, i.e.,

$$I_2 = \{R_1, sol\} \tag{4.15}$$

When the server receives the puzzle solution from the client, it first checks the puzzle expiration time t_{exp} and ignore the client's request if $t_{exp} <$ current time. Then, it verifies that the message integrity code h_s is valid and $hash(sol||h_2) == h'$; the server serves the client request if both conditions are satisfied, and respond with corresponding service response *resp* in message R_2; if either conditions is not satisfied, the server ignores the client's request.

4.5 EVALUATION OF PUZZLE+ DDOS DEFENSE

In this section, we evaluate the effectiveness of Puzzle+ based DDoS defense. We assume the evaluation framework we described in Section 3.3 of Chapter 3 with the following additional details:

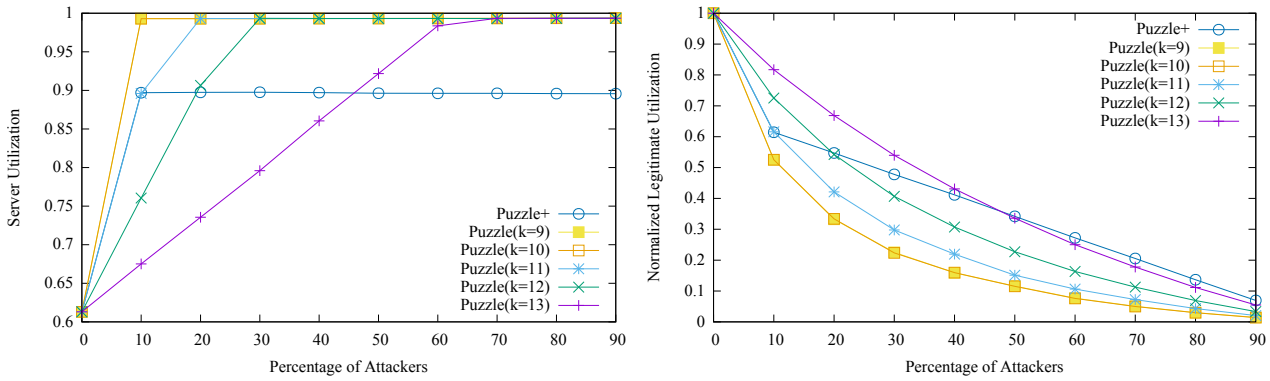
- We use a smaller network topology that consists of a total 342 nodes with 236 client nodes, so that we will be able to run more simulations with different settings in the same amount of physical time. Since the protected resource in our evaluations is the server's computing power but not the bandwidth or latency, it is safe to say that the results of our evaluation were not effected by the type or scale of the network topology we choose.
- Two attack types are considered: the *puzzle resisting attack* and the *replay attack*. Although flooding attacks are also studied, we do not report results on them. As both the existing computational puzzles or Puzzle+ can trivially reject service to almost all flooding requests since they do not have accompanying puzzle solution, the results look as if the server is not effected by the attack at lot, hence are less interesting.

- The attacker in puzzle a resisting attack tries to solve as many puzzles as fast as it can to generate huge volumes of requests with valid puzzle solution. The malicious clients that use puzzle resisting attack is implemented to solve puzzles non-stop to accumulate as many puzzle solution as possible.
- The attacker in puzzle replay attack tries to reuse a puzzle solution at a preconfigured rate until the puzzle expiration time.
- For all simulations that uses Puzzle+, we set the target utilization $\rho*$ in the puzzle hardness model given by Equation (4.12) to 0.9. What this means is that Puzzle+ will try to keep the server utilization at around 90% at all times. The server's service rate μ is set to the product of the number of clients multiplied by the average request rate per client. This will let the server be utilized somewhat closer to its maximum capacity when there is no attack, so that we can easily simulate the effect that the attack traffic makes the server exceed its capacity. In all our simulations in this chapter, the server is about 61% utilized when there is not attack (meaning all clients are legitimate clients).
- CPU frequencies of all clients are set to the same value in all our simulations in this section. In next section, we relax this restriction to experiment with the case where different clients have different CPU configurations.

4.5.1 Effect of Puzzle Hardness

To be able to measure solely the effect of the puzzle hardness model in Puzzle+, we compare Puzzle+ against the fixed hardness puzzles, where the two mechanisms only differ in the way that they compute the puzzle hardness. Puzzle+ uses the puzzle hardness model in Equation (4.12), whereas the fixed hardness puzzle uses a constant puzzle hardness value for all puzzles throughout the same simulation run. We experiment with DDoS defense effectiveness of using the two different approach for setting puzzle hardness.

Figure 4.6 shows the legitimate utilization and *normalized legitimate utilization* of the server during puzzle resisting attacks. *Normalized legitimate utilization* is obtained by dividing the legitimate utilization by the server's total utilization, i.e., $\tilde{\rho}_{legit} = \frac{\rho_{legit}}{\rho}$. It gives a better view of how much of the server's resources are spent on processing legitimate requests



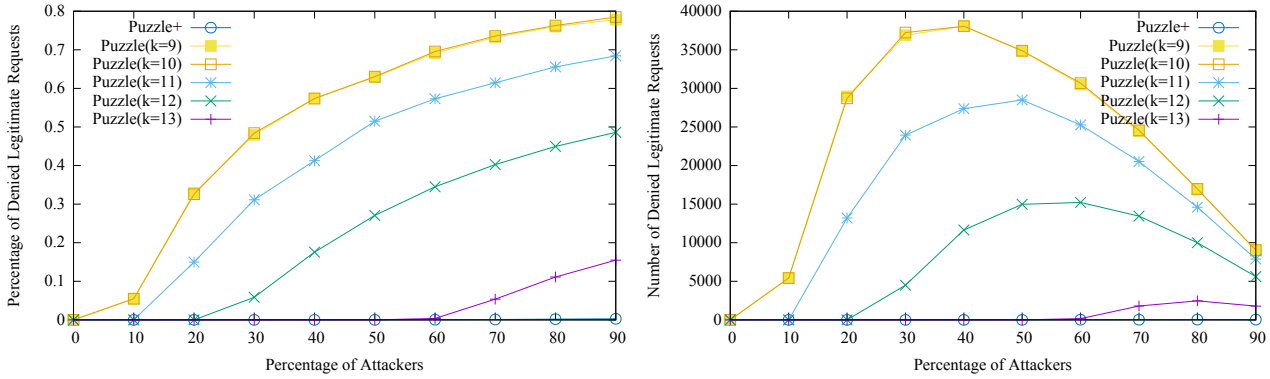
(a) Server utilization

(b) normalized legitimate utilization

Figure 4.6: Utilization and legitimate utilization of the server during puzzle resisting attack

regardless of the total utilization of the server. Lines 'Puzzle(k=9)' to 'Puzzle(k=13)' in Figure 4.6 correspond to the results using fixed puzzle hardness values $9, 10, 11, 12, 13$, and the line 'Puzzle+' corresponds to the results using the puzzle hardness model of Puzzle+. As we can see, using Puzzle+ model provides better legitimate utilization under different attack intensity. Even when 90% of the client population is malicious clients, Puzzle+ model still guarantees a significant portion, that is roughly equal to the percentage of legitimate clients, of the server resources to be allocated to legitimate clients.

Of course, utilization metric alone cannot give the whole picture of how well a model is performing. Figure 4.7 gives the total number and the percentage of denied legitimate requests during the puzzle resisting attacks. Note that the fixed puzzle hardness model can perform really poorly if the puzzle hardness is not correctly selected, as shown in the case where hardness is set to below 13. Even when the hardness is fixed to 13, still a significant percentage of legitimate requests are being denied. One interesting observation can be made by looking at Figure 4.7b is that the total number of denied requested during the attack is decreased as the percentage of malicious clients increased above a certain threshold when using fixed hardness puzzles. This is because as the malicious client population grows, the legitimate client population shrinks, and consequently the total number of requests actually



(a) Percentage of denied legitimate requests

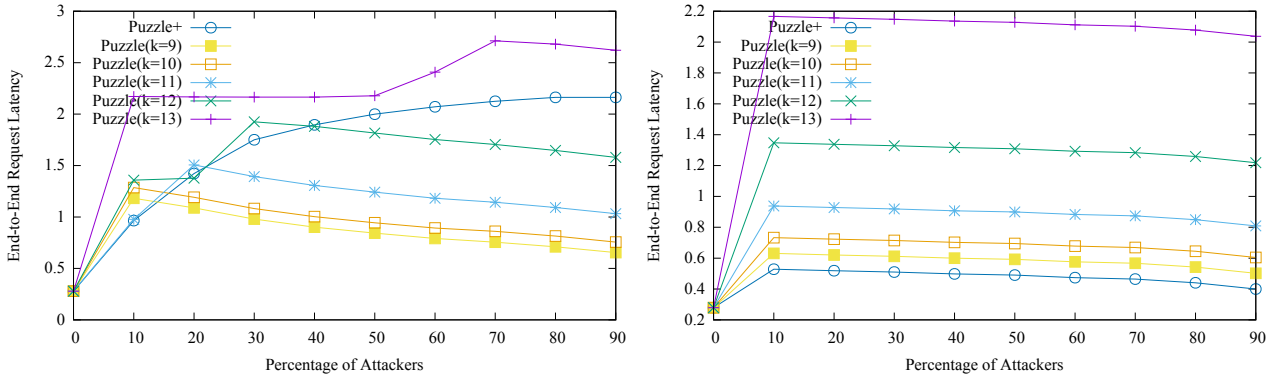
(b) Number of denied legitimate requests

Figure 4.7: Number of denied legitimate requests during puzzle resisting attack

being sent by legitimate clients decreases.

As Figure 4.7a shows, Puzzle+ guarantees almost all legitimate requests being serviced across all attack intensity values, and provides the best protection in comparison. For fixed hardness puzzles, the percentage of denied legitimate requests decreases as the puzzle hardness increases, with the fixed hardness puzzles when $k = 13$ providing the best protection. With the observation of such a trend, one might say continuously increasing the puzzle hardness can further improve the effectiveness of the fixed hardness puzzles. However, doing so would come at the price of continuously worsening request latency, as shown in Figure 4.8a. The fixed hardness puzzle is already gives the worst performance in terms of end-to-end request latency when the puzzle hardness $k = 13$, and increasing the hardness beyond that will further increases the request latency. Puzzle hardness model utilized by Puzzle+ automatically adjusts the hardness as the system load situation changes to provide better protection in terms of percentage of denied requests and latency.

Figure 4.8a compares the request latency of Puzzle+ with fixed hardness puzzles when the server is under flooding attack. Recall that flooding attackers do not solve puzzle and try to overwhelm the server purely by large volumes of request floods. As we mentioned, such request floods can easily be defended by most puzzle protocols simply by ignoring



(a) puzzle resisting attack

(b) flooding attack

Figure 4.8: Average end-to-end request latency

all requests that do not have accompanying puzzle solutions. As such, setting the puzzle hardness to a small value would be enough to counter such attacks, and significantly decreases the request latency. As shown in Figure 4.8a, Puzzle+ automatically adjusts the puzzle difficulty to a smaller value to minimize the request latency, and provides better service quality in comparison with the fixed hardness puzzles. The reason Puzzle+ can adjust down puzzle difficulty is because the server utilization goes down as all the flooding requests being filtered out, the puzzle hardness model given in Equation (4.12) adjusts down the puzzle difficulty when the server utilization is down.

In summary, experiment results show that Puzzle+ combines the lower request latency advantage of easier puzzles with the higher legitimate utilization benefit of harder puzzles, and strikes a good latency utilization tradeoff. Moreover, the percentage of legitimate requests being denied at the server is always the lowest for Puzzle+ in comparison with all different configuration of fixed hardness puzzles.

4.5.2 Replay Attack Prevention

To measure the impact of replay attacks and the effectiveness of the Auto-Expire Cache based replay prevention technique proposed in Section 4.2, we compare Puzzle+, which uses

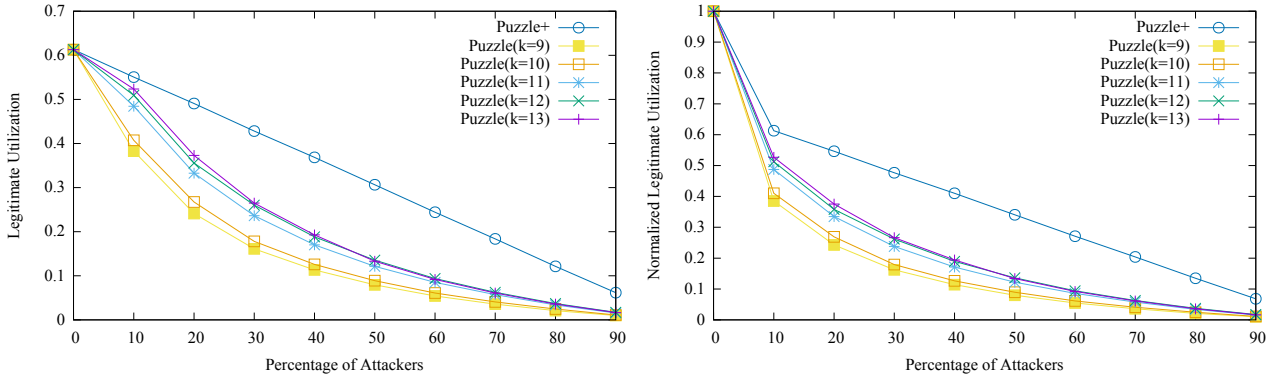
Auto-Expire Cache based replay prevention, against the fixed length puzzle scheme that we introduced previously.

The puzzle solution retaining duration Δ in the Auto-Expire Cache is set to 3 seconds. This is sufficient for all simulations since the hardest puzzle that Puzzle+ utilizes does not take more than 3 seconds. Our simulations show that the average size of the Auto-Expire Cache with the 3 second retention period is around 350, which incurs a negligible memory cost on the server.

In our study, the puzzle solution replay attack is modeled as following: the attacker saves the puzzle solution and other required information for replaying attack in a separate data structure that we call *token chest*. We call each entry in the token chest a *token*. Tokens are stored in the token chest in the order of *the earliest expiring first*, which means that tokens whose corresponding puzzles are expiring earlier will be used earlier to get the most of the same set of tokens.

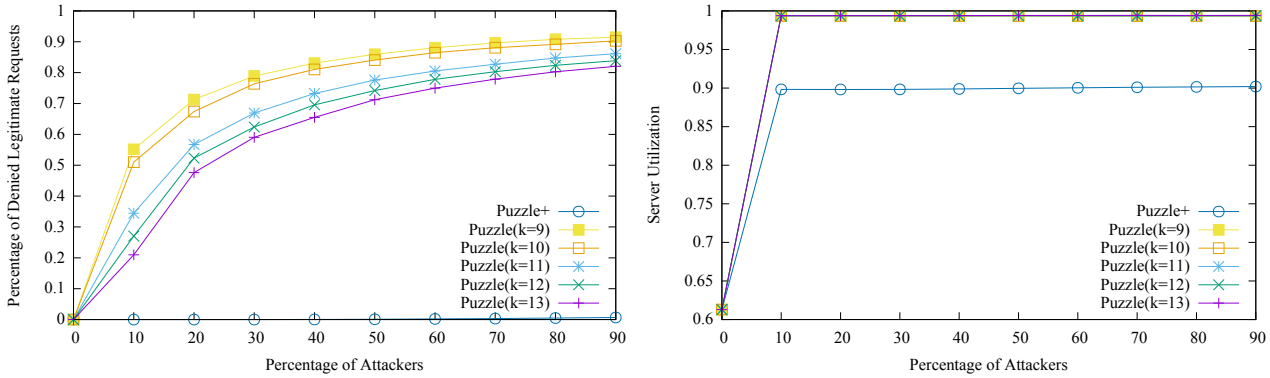
Recall that the malicious clients send requests at 10 times the rate of the legitimate clients in our simulation experiments. In a non-puzzle-replay attack setting, a malicious client's rate of sending service-eligible requests is controlled by the puzzle hardness, hence its effective request rate goes far below the 10 times that of the legitimate client's rate. But, in the replay attack setting, if a malicious client can reuse the same puzzle solution many times until they expire, it can achieve the 10 times rate or a rate that is close to 10 times. So, even before looking at the experiment result, one can imagine that a puzzle scheme without the ability to effectively prevent replay attacks will greatly limit its effectiveness against DDoS attacks that utilize replay.

Figure 4.9 shows the utilization of the legitimate utilization drops drastically, as expected, when fixed hardness puzzles that do not deploy the Auto-Expire Cache based replay prevention are used. The same outcome applies to all different configuration of puzzle hardness $k = 9, 10, ..., 13$. In the case of Puzzle+ which uses the Auto-Expire Cache based replay prevention mechanism, the legitimate clients are still guaranteed a fraction of the server's capacity in proportion to the size of the legitimate client population. The legitimate utilization values for Puzzle+ in the replay attack case (Figure 4.9) are almost identical to the legitimate utilization values for Puzzle+ in the puzzle resisting attack case (Figure 4.6).



(a) legitimate utilization (b) normalized legitimate utilization

Figure 4.9: Legitimate utilization of the server during the replay attack



(a) Percentage of denied legitimate requests (b) Server utilization

Figure 4.10: Percentage of denied legitimate requests and server utilization during replay attacks

Figure 4.10a compares the percentage of denied legitimate requests when Puzzle+ is used versus the fixed hardness puzzle is used. Even with harder puzzles, such as when hardness $k = 13$, the majority of the legitimate requests are being denied under almost all configurations of the attack intensity when the fixed hardness puzzles are used. In contrast, Puzzle+ effectively prevents replay attacks in all circumstances, and guarantees the same level of protection that it did under puzzle resisting attacks.

Another advantage of Puzzle+ that is worth mentioning is that it guarantees the server utilization to be around the target utilization value $\rho*$ under most circumstances. This is illustrated by the experimented results in Figure 4.10b.

The results here combined with the results in Section 4.5.1 show that Puzzle+ provides strong defense against all three types of DDoS attacks — flooding attacks, puzzle resisting attacks, and replay attacks — that we study.

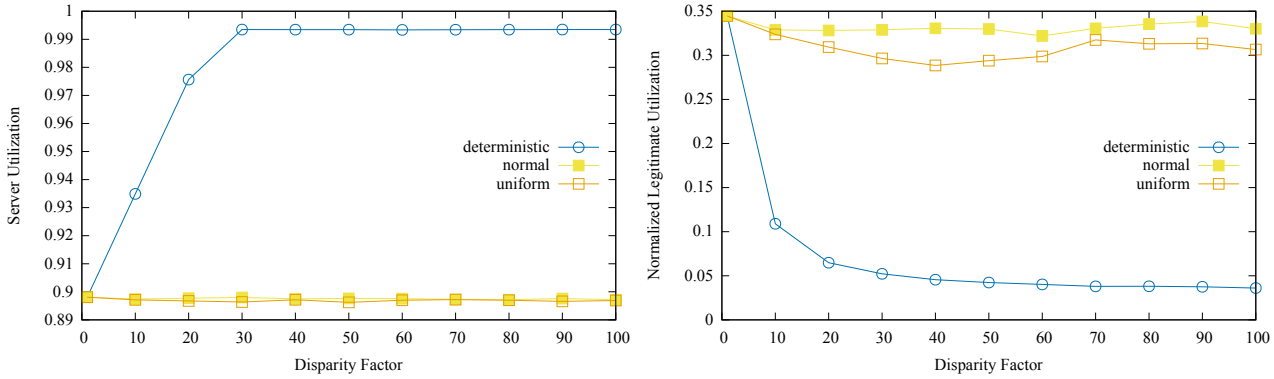
4.6 EFFECT OF DISPARITY IN CLIENT COMPUTATIONAL POWERS

All of our experiments in the previous section assume that all clients machines have the same CPU frequencies. In this section, we study the impact of the variance in the client machine CPU frequencies on the DDoS efficacy of computational puzzles. We define a metric called *disparity factor* to measure the variance in the computational resources that are available to all clients.

Definition 4.1. Disparity factor *is the ratio of the CPU frequencies of the most powerful and the least powerful client machines out of all client machine population.*

The disparity factor alone cannot describe the client machine CPU frequency values. We also need to know what probability distribution the CPU frequency values follow. We consider three different probability distributions: normal distribution, uniform distribution, and deterministic distribution. The last one — deterministic distribution — is a degenerate distribution where a random variable always takes a single value. We adopt the following methods to generate CPU frequency values that follow different distributions:

- When using normal distribution to generate CPU frequency values, we do not discriminate for which type of client we are generating the CPU frequency for. That means malicious client machines will have mixture of high, low, and average CPU frequencies, and so do legitimate client machines. However, the ratio of the highest and the lowest CPU frequency values will approximately be equal to the given disparity factor. The generated CPU frequency values will, of course, form a normal distribution, and fall



(a) server utilization (b) normalized legitimate utilization

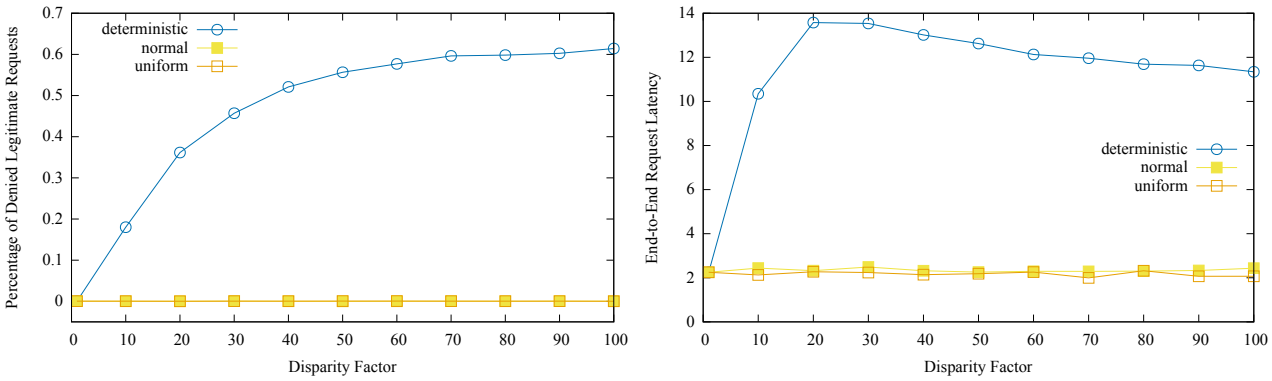
Figure 4.11: Impact of disparity factor on the utilization of the server

within a range whose length results in the disparity factor with a probability of 0.995. A mean CPU frequency value of $1,000,000$ is used for normal distribution.

- The method for generating frequency numbers using the uniform distribution is similar, except the generated numbers will fall within a range whose length equal to the disparity factor almost surely (meaning the probability is 1.0). It is *almost surely*, because uniform distribution takes a lower and upper bound, and we can set these values so that $upper : lower = disparity\ factor$. As in normal distribution case, a mean CPU frequency value of $1,000,000$ is used for the uniform distribution.
- In the deterministic distribution case, we discriminate between the legitimate clients and malicious clients. Specifically, legitimate client CPU frequencies are always set to a fixed value f_g, and malicious client CPU frequencies are always set to $disparity\ factor \times f_g$. We fix the value of f_g to be $f_g = 1,000,000$.

For all of our experiments in this section, we fix the percentage of malicious clients to 50%. The disparity factor is varied from 1 (no disparity) to 100. Puzzle+ is used as the defense mechanism in all experiments. Results are shown in Figure 4.11 and 4.12.

First of all, Puzzle+ still performed very well against DDoS attacks across all metrics, regardless of how big the disparity factor is, when client CPU frequencies are taken from



(a) Percentage of denied legitimate requests

(b) Request latency

Figure 4.12: Impact of disparity factor on latency and denial rate of legitimate requests

normal distribution and uniform distribution. Recall that clients are not discriminated when their CPU frequency values are taken from normal and uniform distributions. So, if we assume that malicious client machines are regular desktop and laptop computers that are turned into zombies or bots, then Puzzle+ can still be a very effective defense against DDoS.

However, if we assume even stronger attacks where the attacker is using only the best machines that are 10 or even 100 times more powerful than average client machines, then Puzzle+ becomes less effective. This can be seen by looking at the experiment results in Figure 4.11 and 4.12 for the case where client CPU frequencies follow "deterministic" distribution. Recall that malicious client CPU frequencies are set to *disparity factor* $\times f_g$ when using deterministic distribution, where legitimate client CPU frequencies are set to f_g. So, for *disparity factor* $= 100$, all malicious client machines will be 100 times more powerful than all legitimate client machines.

When the malicious clients all have such powerful machines, Puzzle+ becomes less able to control the server utilization, and the utilization goes well over the target utilization of 0.9 as shown by line "deterministic" as shown in Figure 4.11a. The reason for this is because one of the factors in the puzzle hardness model in Equation (4.2) and (4.12), that are used

by Puzzle+, is average CPU frequency f. With malicious client machines getting 10 to 100 times CPU frequencies that of the average legitimate clients', the average frequency f that the server becomes far below the true average. One way to fix this would be just increase the average value, but there must be systematic way of increasing it and stopping the increase.

4.7 CONCLUSION

In this chapter, we first showed the limitations of existing computational puzzle schemes: (1) lack of a mathematical model for determining puzzle hardness on a per request basis; (2) susceptibility to puzzle solution replay attacks; and (3) lack of systematic approach for deciding when to switch puzzle on or off. We then provided novel solutions to each of these problems, and incorporating these solutions into a single framework — Puzzle+. We study the goodness of the solutions we proposed by experimenting with flooding, puzzle resisting, and replay attacks against Puzzle+ and by comparing the results to that of the existing puzzle schemes. Our results show that Puzzle+ can provide strong protection against all three types of DDoS attacks when it is assumed that all clients have the same computational powers. Puzzle+ was also able to provide a strong protection against these attacks when there exists large disparity among the computational powers of clients, but became significantly less effective when all malicious client machines are configured to be 10 - 100 times more powerful than any legitimate client machine.

Another limitation of Puzzle+, and all computational puzzle frameworks for that matter, is that they require clients to perform expensive computational tasks. More importantly, results of such puzzle computations do not contribute towards solving meaningful problems that provide utility to end-users or other entities.

In next two chapters, we try to eliminate or alleviate the limitations of Puzzle+ and other computational puzzles by not requiring the clients to solve hard computational puzzles or leveraging the puzzle computation work towards solving meaningful problems to make puzzle based DDoS defense more effective and attractive for practical use.

5.0 PRODUCTIVE PUZZLE FRAMEWORK

Although, puzzle based DDoS mitigation methods are promising, they require expensive computations to be performed on client machines and the client resources spent on computing puzzles amount to nothing except for verifying that the client spent them. More specifically, the puzzle computations do not result in solutions to meaningful problems, nor do they accomplish a meaningful task that can benefit users in some way. The most commonly given task in puzzle frameworks is computing millions of cryptographic hashes, and such computations do not constitute a meaningful functionality or service that provides utility to some entity. One may say that current puzzle frameworks are wasteful of the client resources, and we called this problem *wasteful computation problem* in Chapter 6.

In this chapter, we propose a novel computational puzzle idea, called *productive puzzles*, and aim to tackle the wasteful computation problem. The work to be completed in a productive puzzle is not hash reversal computations or other type of repetitive cryptographic operations. Instead, they can be tasks from applications or services that provide meaningful functionalities to users, although cryptographic operations can also be used. We build a productive puzzle protocol around the idea of productive puzzles, and enhance it with other novel ideas for computing puzzle hardness and preventing solution replay. The main contributions of this chapter includes:

- Introduction of the novel idea of productive puzzles, and design of the productive puzzle framework to address the wasteful computation problem;
- Design of mechanisms to detect cheating during the productive puzzle verification, and provision of a mathematically proven upper bound on the probability of successfully cheating;

- Study of DDoS effectiveness of productive puzzles in a realistic experimental environment.

5.1 PRODUCTIVE PUZZLES

In this section, we introduce the productive puzzles concept and describe the design of the productive puzzle framework.

5.1.1 Overview

Existing methods of constructing puzzles focus on designing a function F such that computing $y = F(x)$ is easy, but computing the inverse $x = F^{-1}(y)$ is *moderately hard*, where the hardness is measured in terms of time complexity (CPU bound), space complexity (memory bound), or both. Given such a function F and the output y, the client must spend significant amount of resources—whether that be time, space, or both—to compute the input x and prove to the server the validity of x before the server starts fulfilling its request. The server can verify the validity of x with ease since computing $y = F(x)$ is trivial. Some constructs may use slightly different function pairs $G(x)$ and $F(x, y)$, where $F(x, y) \in \{0, 1\}$ and computing $F(x, y)$ is trivial relative to computing $G(x)$. Commonly used functions in existing puzzle constructs are cryptographic hash functions, as in *hash reversal* [35] and *hash trail* [5] puzzle constructs, or other cryptographic functions that display *one-wayness*—easy to compute on input, but hard to invert given the output. In all existing puzzle schemes that use such constructs, the work performed by the client is merely for proving to the server that it has spent its resource, and it does not benefit any involved party in any meaningful way. In short, existing puzzle schemes are wasteful of client resources.

In productive puzzles, the puzzle is constructed using a set of inputs $x_1, x_2, ..., x_k$ and a set of functions $F_1, F_2, ..., F_k$, and the solution is consisted of a set of outputs $y_1, y_2, ..., y_k$, where $y_1 = F_1(x_1), y_2 = F_2(x_2), ..., y_k = F_k(x_k)$, and F_i can be any function for $1 \leq i \leq k$. The server pre-computes $y_1, ..., y_j$, where $j < k$, and it does not have the remaining outputs $y_{j+1}, ..., y_k$. The server then randomly permutes the $< input\ value, function >$ pairs to get

the final puzzle puz, i.e.,

$$puz = rand\_permute(< x_1, F_1 >, < x_2, F_2 >, ..., < x_k, F_k >). \tag{5.1}$$

When verifying a puzzle solution $sol = (y'_1, y'_2, ..., y'_k)$ submitted by the puzzle solver, the server will simply verifies that $y_1 = y'_1, y_2 = y'_2, ..., y_j = y'_j$, i.e., the server verifies the correctness of the subset of tasks that it has already precomputed.

To describe productive puzzles in a more plain language, a productive puzzle consists of a set of tasks, where the server has precomputed solutions for a subset of them. Here, a *task* is defined as a unit of execution that takes some input x and produces some output y. A task for which the server knows the output is called a *known task*, and a task for which the server does not have the output is called an *unknown task*.

Assuming the client does not know which tasks are known tasks and which are unknown tasks, it must compute all tasks indiscriminately. A malicious client can try to cheat by not computing a random subset of the tasks and providing fake solutions to them, but it does so at the risk of being discovered with a certain probability. By having solutions to a large enough subsets of the tasks, one can discover cheating with very high probability.

Any task that has not been seen by a client can be used in the construction of a productive puzzle for that client. In some cases, the server's own tasks can be used in the puzzle construction, thus help decreasing the load on the server when necessary. We call a server *thin server* if it offloads the computation of its tasks to its clients using productive puzzles, and productive puzzles in such use use are referred to as *pro-server puzzles*. In thin server mode, the server's ability to process requests will no longer be susceptible to DoS attacks, because the more an adversary tries to saturate server's capacity, the more pro-server puzzles it has to solve, and offloading more tasks of the server to itself. However, not all servers can become thin servers, because often it is impossible to offload the server's tasks to the client without leaking sensitive information. Nevertheless, there can be some applications that can make use of the thin server mode.

5.1.2 Probability of Cheating

Now, let us take a closer look at the probability of successful cheating in productive puzzles.

Definition 5.1. *Assuming a productive puzzle puz is consists of p known tasks and u unknown tasks, a client is considered cheating if it submits the puzzle solution by completing only w tasks, for any $w < (p + u)$.*

Informally, cheating is not doing some of the tasks in a productive puzzle.

Definition 5.2. *A successful cheating is defined as being able to form a puzzle solution that can pass the server's verification while cheating.*

We assume that an adversary does not have any information regarding which tasks are known tasks. To avoid computing all $p + u$ tasks, the adversary can choose w of them to compute, where $w < p + u$. The w tasks it choose must contain all p tasks that are known to the server, otherwise the server will detect the cheating with 100% certainty. In other words, w must satisfy the inequality $w \geq p$. There are $p + u$ choose w or $\binom{p+u}{w}$ many ways of choosing w tasks to compute out of the $p + u$ total tasks. The combinations that can pass the server's verification must contain all p known tasks, thus the valid combination first chooses all p known tasks out of p, i.e., $\binom{p}{p}$. The remaining $w - p$ tasks are chosen from the u unknown tasks, and there are $\binom{u}{w-p}$ many ways. Consequently, there are $\binom{p}{p} \times \binom{u}{w-p}$ many possibilities that includes all p known solution, and thus can pass the server detection. By combining all possible of ways of choosing w tasks out of $p + u$ tasks and the number of ways to choose w out of $p + u$ tasks that can pass the detection, we can get the following probability of successful cheating while completing only w tasks:

$$Pr(w) = \frac{\binom{u}{w-p}}{\binom{p+u}{w}} \tag{5.2}$$

$$= \frac{u!}{(w-p)!(p+u-w)!} \cdot \frac{w!(p+u-w)!}{(p+u)!} \tag{5.3}$$

$$= \frac{u!w!}{(p+u)!(w-p)!}. \tag{5.4}$$

When there are equal number of known and unknown tasks, i.e., $p = u$, Equation (5.4) can be simplified as

$$Pr(w) = \frac{p!w!}{(2p)!(w-p)!}, (\text{when } p = u). \tag{5.5}$$

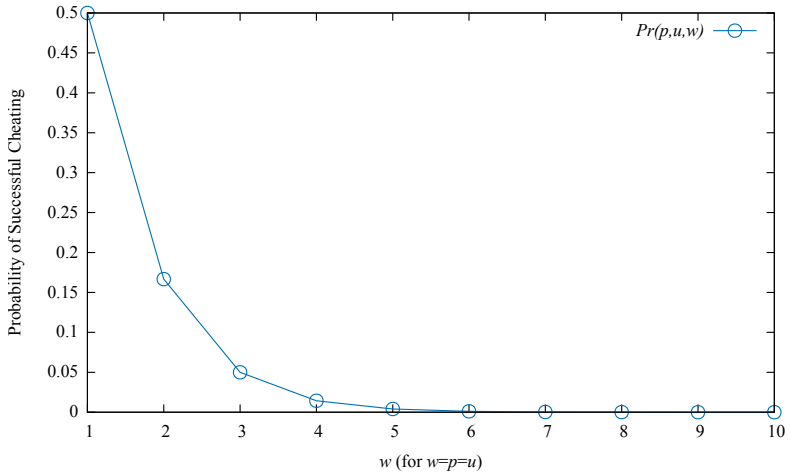


Figure 5.1: Successful cheating probability when w varies from 1 to 10, for $w = p = u$.

Probability of successful cheating when the number of known and unknown tasks p, u, and the number of tasks w completed by the adversary are all equal, i.e., $w = p = u$, is given in Figure 5.1. If the adversary chooses to compute only 3 tasks when there are 3 known and 3 unknown tasks, the probability of it successfully choosing the set tasks that can pass the server's verification is 5%, or the server can detect such cheating with 95% probability. As the number of known and unknown tasks increases, the probability of an adversary cheating successfully decreases exponentially. We can establish a upper bound for successful cheating in the following theorem.

Theorem 5.1. *If an adversary has no non-negligible knowledge regarding the known tasks in a productive puzzle, then the upper bound on the probability of successful cheating* Pr_{max} *is* $\frac{u}{p+u}$ *for any configuration of productive puzzles.*

Proof. The $\Pr(w)$ increases as w increases according to Equation (5.4), and the maximum w that satisfy the inequality $w < (p + u)$ from Definition 5.1 is $w = p + u - 1$. Thus, we can

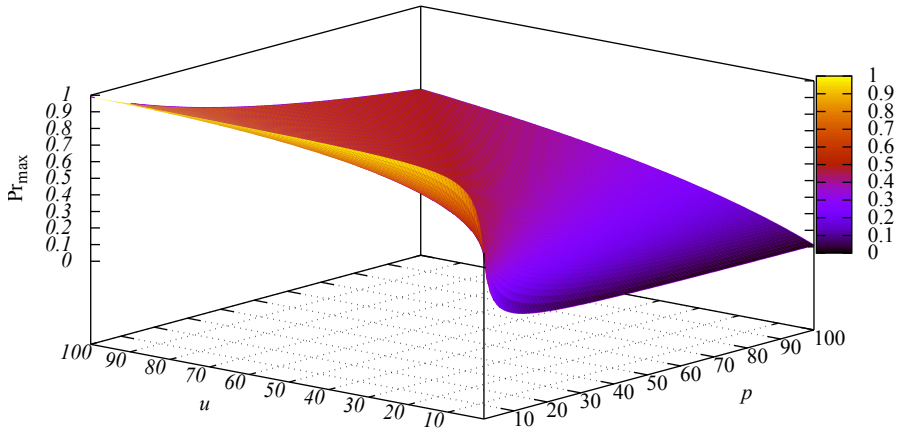


Figure 5.2: Successful cheating upper-bound when using *known-unknown test*

get Pr_{max} by taking $w = p + u - 1$ into Equation (5.4)

$$\begin{aligned} Pr_{max} =& \frac{u!(p+u-1)!}{(p+u)!(p+u-1-p)!} \\ =& \frac{u}{p+u}. \end{aligned} \tag{5.6}$$

□

The upper bound $Pr_{max} = \frac{u}{u+u} = 0.5$ when $p = u$. Figure 5.2 shows the maximum cheating probability when both p and u are varied from 1 to 100.

5.1.3 Honesty Test

In the context of productive puzzles, an *honesty test* is defined as a test that is devised to detect cheating. We already described one kind of honesty test where the subjects are given a mixture of known and unknown tasks and subsequently judged by their solutions to the known tasks. We call this honesty test a *known-unknown test*. There can be other honesty tests that are different from it.

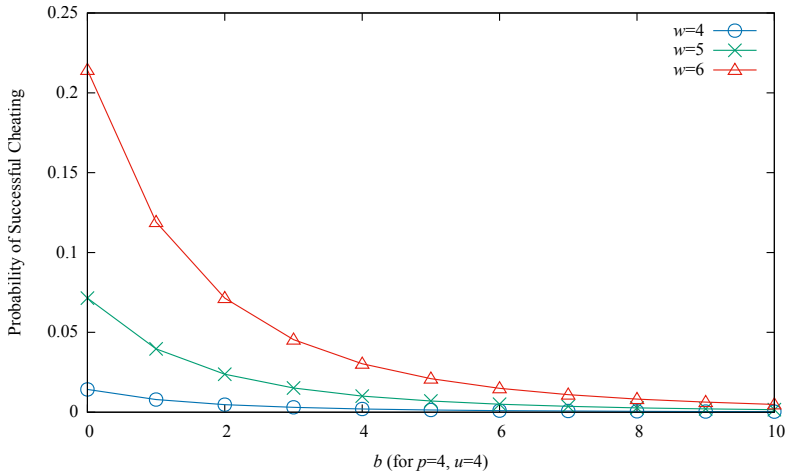


Figure 5.3: Effect of using bogus tasks on the probability of successful cheating

We can improve the known-unknown test slightly by adding a third type of tasks, that we call *bogus tasks*. A *bogus task* is a randomly generated task that resembles a real task, and is hard to be differentiated from a real task. We call the honesty test that uses known, unknown, and bogus task combination a *knonw-unknow-bogus test*. The motivation for using a known-unknown-bogus test is because it improves upon known-unknown test in terms of decreasing the probability of successful cheating. Let us illustrate this by deriving the probability of successful cheating when using bogus tasks.

As we include bogus tasks in a productive puzzle, the probability of successful cheating in Equation (5.4) becomes as following:

$$Pr(w) = \frac{(u+b)!w!}{(p+u+b)!(w-p)!}, \tag{5.7}$$

where, b is the number of bogus tasks. When p and u are fixed, increasing b will decrease the value of $Pr(w)$, this is illustrated in Figure 5.3 for $p = 4$ and $u = 4$.

In addition to lowering the probability of successful cheating, bogus tasks can be used to replace the unknown tasks when there are enough unknown tasks in the system. The shortage of unknown tasks in the system. Another approach that can used where there is shortage of unknown tasks or known tasks is to fall back to using the traditional hash based

puzzles. In this way, the server does not have to face the problem of not having enough puzzles to send to clients when it is under attack.

The specific algorithm for generating a bogus task can vary depending on the type of the task being used by a system. For example, if the real task is counting the number of times each word appears in a piece of text, then a bogus task can be performing the same operation on a synthesized piece of text or a text copied from a random page on the Internet. A bogus task can be constructed by using one of three approaches: (1) using the same task function as the real task, but a synthesized input; (2) using the same task input as a real task, but a synthesized or modified task function; or (3) using both synthesized task function and task input.

5.1.4 Fault-Tolerance through Voting

To further minimize the upper-bound on successful cheating and minimize the chance of an erroneous solution being accepted, a task can be distributed multiple times in multiple productive puzzles. Some type of voting mechanism can then be used to choose the correct solution of the task. We adopt *majority voting* — one of the most commonly used voting mechanisms.

We define an *m-majority voting* mechanism to be one that requires the winning solution to have at least m votes. We call m the *majority threshold*, and it represents the minimum vote necessary to win the election. Assuming the solution of a task is either *correct* or *incorrect* and assuming that the incorrect solutions match, then at most $2m - 1$ copies a task need to be distributed in order to get at least m matching solutions for it. Note that it is reasonable to assume that the incorrect solutions match, because an adversary will dilute its votes, hence lower the chance of its incorrect solution being accepted, if it provides unmatching incorrect solutions.

Assuming that ε is the probability of an individual solution being incorrect, then the probability of exactly d solutions being incorrect is given by the following:

$$Pr\{\text{exactly } d \text{ incorrect solutions}\} = \binom{2m-1}{d} \varepsilon^d (1-\varepsilon)^{2m-1-d} \tag{5.8}$$

We call ε the *per-solution error rate*.

For the incorrect solution to be accepted as the final solution of the task, there must be at least m incorrect solutions. Thus, the probability $\sigma(\varepsilon, m)$ of a task's final solution being incorrect is given as

$$\begin{aligned}\sigma(\varepsilon, m) =& Pr\{\text{accepting incorrect solution}\} \\ =& Pr\{d \geq m\} \\ =& \sum_{d=m}^{2m-1} \binom{2m-1}{d} \varepsilon^d (1-\varepsilon)^{2m-1-d}\end{aligned} \tag{5.9}$$

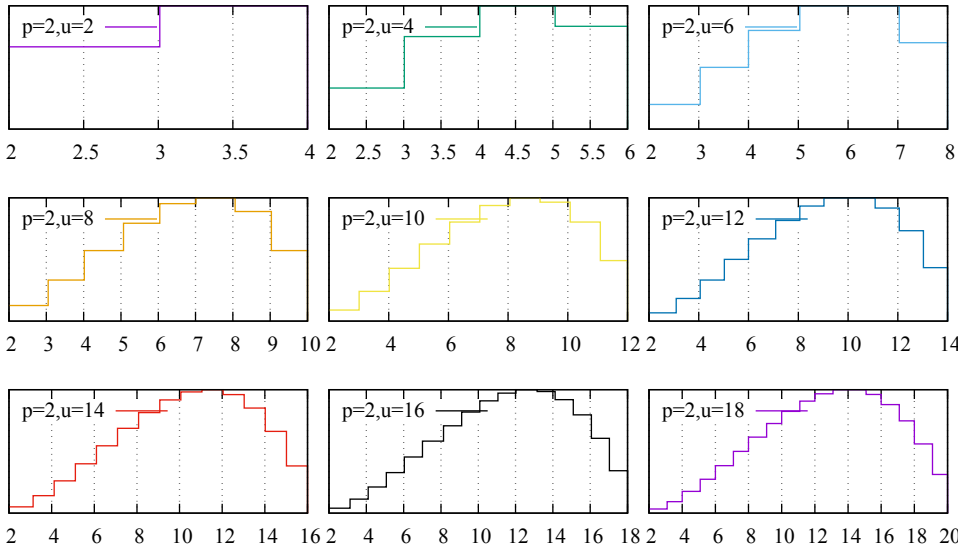
We call $\sigma(\varepsilon, m)$, the *per-task error rate*.

Now, let us try to compute the per-solution error rate when voting is combined with the *known-unknown test* that we described earlier. In this combination, an individual solution must first pass the test, then being accepted by the voting. So, the per-solution error rate ε becomes equivalent to the probability of an individual solution from a malicious client passing the test, which is given by the following equation:

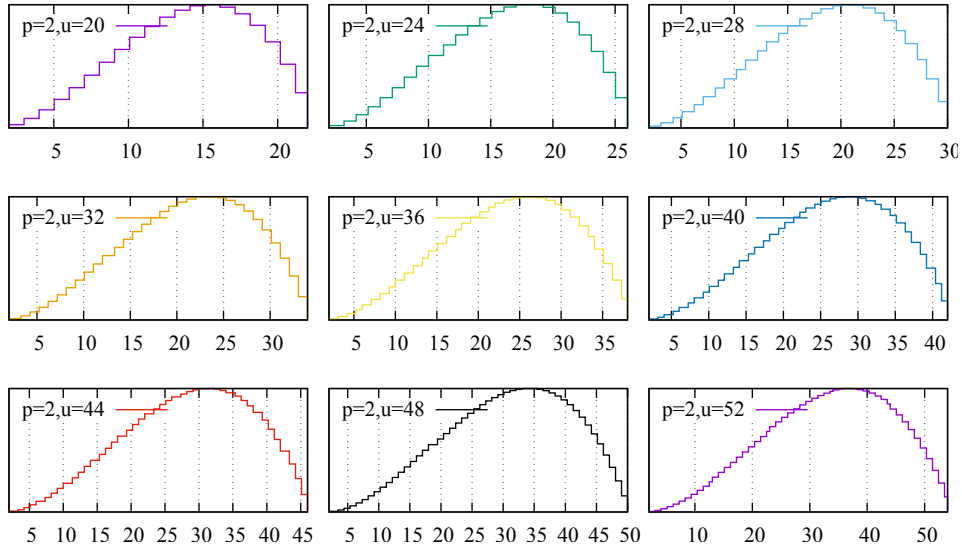
$$\begin{aligned}\varepsilon(p, u, w, r_b) =& Pr\{\text{malicious client} \cap \text{incorrect solution} \cap \text{passes test}\} \\ =& r_b \cdot \frac{p+u-w}{p+u} \cdot \frac{u!w!}{(p+u)!(w-p)!},\end{aligned} \tag{5.10}$$

where, r_b is the fraction of the client population that are malicious, $\frac{p+u-w}{p+u}$ is the fraction of the solutions that are incorrect in a puzzle computed by a malicious client, and the last terms is given by Equation (5.4), it is the probability of a puzzle solution computed by a malicious client passing the known-unknown test. Figure 5.4 plots the per-solution error ε based on Equation 5.10 for the case when $p = 2$.

An upper-bound on the $\varepsilon(p, u, w, r_b)$ can be given as follows. If an adversary has no non-negligible knowledge regarding the known tasks in a productive puzzle, then the upper bound ε_{max} on the per-solution error rate is given by $\frac{r_b u}{(p+u)^2}$ when $p = u$. The value of ε in Equation (5.10) increases as the value of w increases. This is because the exponential increase given by $w!$ in the last term is far larger than the additive decrease given by $-w$ in the second term. In order to get the maximum value of ε, w should take its maximum



(a) $u = 2, 4, ..., 18$



(b) $u = 20, 22, ..., 52$

Figure 5.4: Per-solution error rate ε for various values of u when $p = 2$

value $w = p + u - 1$ that satisfies the inequality $w < (p + u)$ from Definition 5.1. By taking

$w = p + u - 1$ into Equation (5.10), we get

$$\begin{aligned}\varepsilon_{max}(p, u, r_b) =& r_b \cdot \frac{p + u - (p + u - 1)}{p + u} \cdot \frac{u!(p + u - 1)!}{(p + u)!(p + u - 1 - p)!} \\ =& \frac{r_b u}{(p + u)^2}. \end{aligned} \tag{5.11}$$

By taking the maximum per-solution error rate ε_{max} from Equation (5.11) into the per-task error rate equation in Equation (5.9), we can get the upper-bound $\sigma_{max}(p, u, r_b, m)$ on the per-task error rate as follows.

$$\sigma_{max}(p, u, r_b, m) = \sum_{d=m}^{2m-1} \binom{2m-1}{d} \left(\frac{r_b u}{(p + u)^2}\right)^d \left(1 - \frac{r_b u}{(p + u)^2}\right)^{2m-1-d} \tag{5.12}$$

5.2 PRODUCTIVE PUZZLES FRAMEWORK

In this section, we first describe overall architecture of a system that utilizes productive puzzles, and explain how the productive puzzle framework fits together with the rest of the system. Then, we introduce the *Productive Puzzle Protocol* for the client-server interaction, followed by the description of mathematical models and heuristics for determining the number of tasks in a productive puzzle.

5.2.1 Overall Architecture

We envision the overall architecture of the system that deploys productive puzzles to be approximately look like the one given in Figure 5.5. There are four entities in this system:

- The **client** and the **protected server** are the client and the server that are already defined;
- **Beneficiary** is any entity that benefit from the computational platform that the productive puzzle system provides, and it periodically uploads the tasks to be used in productive puzzles and downloads the solutions that have been verified;
- Lastly, the **productive puzzle system**, which provides the productive puzzle based DDoS protection to the server and a task computation platform to the beneficiary.

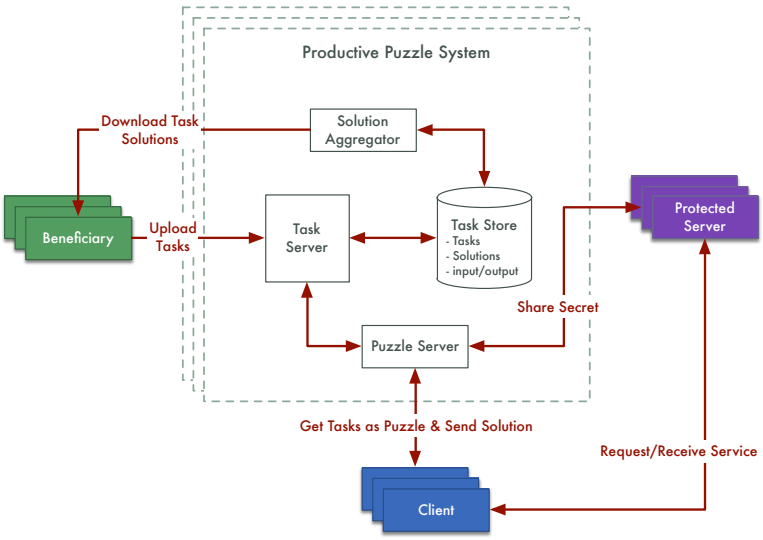


Figure 5.5: Overall architecture of a productive puzzle system

It is preferable to have multiple beneficiaries supplying tasks to the productive puzzle system, in order to form a steady stream of new incoming tasks to be used as puzzles. However, it is possible to have time periods where tasks are *over-supplied* or *under-supplied*. *Over supply* means the productive puzzle system is receiving more tasks than it can store and process, whereas *under-supply* means there are not enough new incoming tasks to be used as puzzles. The phrase *new tasks* implies that there is also a corresponding concept of *old tasks*. A task is called stale task if it has been used more than R_{max} times by productive puzzles; and similarly a task is called *fresh task* if it has been used less than the R_{max} threshold. There are good reasons for not using stale tasks in puzzles. It leads to situations where the client has already seen the task and spends no effort solving it, thereby circumventing the computational delay that is designed to be enforced by the solving the puzzle.

Both over-supply and under-supply problems are fairly easy to deal with. For over-supply, a simple rate-limiting mechanism can be used limit the new task arrival rate. On the other hand, under-supply can be dealt with by falling back to using traditional hash reversal puzzles or mixing hash-reversal tasks into the productive puzzles.

As the diagram in Figure 5.5 shows, the productive puzzle system has four main components that are given as follows.

- **Puzzle Server** is responsible for constructing puzzles and verifying their solutions, which includes applying the known-unknown test on the puzzle solution. It determines the number of tasks k and the number of known and unknowns tasks p and u by using the models that will be introduced in Sections 5.2.3 and 5.2.4, and gathers the tasks needed by the productive puzzles.
- **Task Server** accepts the incoming new tasks uploaded by the beneficiaries, and persists them into the *task store*; meanwhile, it handles requests from the puzzle server and supplies it with the required numbers of known and unknown tasks.
- **Task Store** provides an efficient storage and retrieval of tasks, task input data, and task solutions.
- **Solution Aggregator** determines the final solutions of already solved tasks by applying the voting algorithm in use, and pushes the finalized task solutions to the beneficiaries.

Note that the functionalities needed by the productive puzzle system components are fairly simple, making the entire productive puzzle system fairly lightweight. Meanwhile, it can easily be replicated to have many instances that work entirely independent from each other, thereby leading to a highly scalable and robust system.

We do not describe each component of the system in more details, as they can be implemented using well-known techniques and algorithms. In the remainder of this section, we focus on designing the protocol that governs the communication between the productive puzzle system and the clients, and deriving models for determining what types and how many tasks to include a productive puzzle.

5.2.2 Productive Puzzle Protocol

Based the foundational concepts of productive puzzles introduced in earlier Sections 5.1.1 and 5.1.2, we design a protocol called *Productive Puzzle Protocol (PPP)* to govern the interaction between the server and its clients.

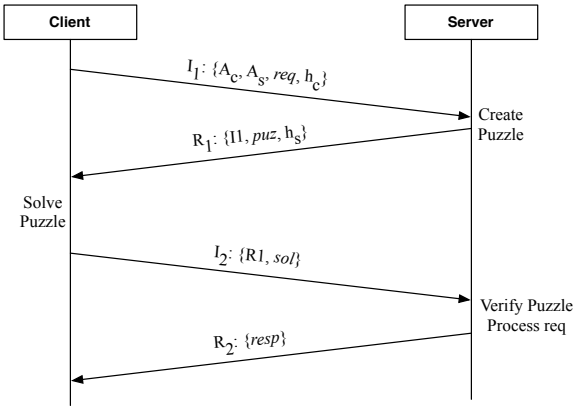


Figure 5.6: Client server interaction in the productive puzzle protocol

Similar to most puzzle protocols, a transaction in productive puzzle protocol consists of four message exchanges: (1) initial client request (I_1), (2) initial server response (R_1), which includes a puzzle, (3) reinforced client request (I_2) that contains a puzzle solution, and (4) final server response (R_2). Figure 5.6 illustrates the client server interaction in the productive puzzle protocol. This four message model assumes that the puzzle server and the actual protected server are the same entity from the clients' perspective, and it does not mean they need to be deployed on the same physical server or serving infrastructure. The protocol can be easily extended to a *redirect model* where the clients are required to talk to two different server entities in a single transaction.

In a redirect model the exchanged messages in a single transaction include: (1) initial client request (I_1), (2) initial server response (R_1), which is a redirect message directing the client to the puzzle server, (3) initial puzzle request (IP_1) to the puzzle server, (4) a puzzle response (RP_1) from the puzzle server, (5) a puzzle solution response ((IP_2), (6) a service token (RP_2), (7) reinforced client request (I_2) that contains a service token, and (8) final server response (R_2). However, we only present the simpler four message model in order not to distract attention away from the important message exchanges in the productive puzzle protocol.

The client's initial request message I_1 is consists the following:

$$I_1 = \{A_c, A_s, req, h_c\}, \tag{5.13}$$

where, A_c and A_s are unique identifiers of the client and the server, such as IP address or URL; req is the client's service request, for example a URL of a Web resource; h_c is an hash-based message authentication code (HMAC) [36] of the previous 3 items, and it is computed as follows.

$$h_c = hmac(K_c,\ (A_c||A_s||req)), \tag{5.14}$$

where, K_c is HMAC secret key of the client, and $||$ is the string concatenation operator. h_c keeps the integrity of the client's message and also plays the role of a client *nonce*.

Next, the server replies with the initial response message R_1, which contains

$$R_1 = \{I_1, puz, h_s\}, \tag{5.15}$$

where, h_s is the HMAC of first two items and computed as

$$h_s = hmac(K_s,\ (I_1||puz)), \tag{5.16}$$

where K_s is the HMAC secret key of the server, puz is the productive puzzle that has the following parts:

$$puz = \{< Tid_1, T_1 >, ..., < Tid_k, T_k >, t_{exp}, token\}, \tag{5.17}$$

where, k is the number of tasks, T_i is the i-th task in the puzzle, Tid_i is the unique task identifier for the task T_i, t_{exp} is the puzzle expiration time after which the puzzle solution is not accepted, $token$ is a 128-bit random string that is used as a unique identifier of the puzzle. The model for computing the number of tasks k is given in the next section (5.2.3). Each task T_i is consisted of the task code Fn_i and input x_i, i.e.,

$$T_i = \{Fn_i, x_i\}. \tag{5.18}$$

Transmitting the code for every puzzle may become too expensive in terms of bandwidth. So, the code could be pushed to the client once when the first request is received, and can be

reused by including only the code unit identifier, $FnID$, in the puzzle. Then, the equation for tasks T_i becomes

$$T_i = \{FnID_i, x_i\} \tag{5.19}$$

After receiving the puzzle, the client computes puzzle solution *sol* by computing each task in the productive puzzle. *sol* is consists of following:

$$sol = \{< Tid_1, y_1 >, ..., < Tid_k, y_k >\}, \tag{5.20}$$

where $y_i = Fn_i(x_i)$.

When the server receives the puzzle solution from the client, it first checks the puzzle expiration time t_{exp} and ignore the client's request if $t_{exp} < current\ time$. The, it iterates over all items in solution *sol* and checks whether y_i is correct only if $Tid_i \in \mathbb{K}$, where $\mathbb{K}$ is the set of task identifiers of all known tasks. If all known tasks are correct, then the server processes the client's request, and sends reply message R_2.

5.2.3 Puzzle Hardness

Recall that the per-request puzzle hardness model $d_{req} = \frac{N(t)f}{\rho^*\mu}\frac{t_{req}}{t_{avg}}AF(t)$ for computational puzzles is given in Equation (4.2) in Section 4.1.1. As productive puzzles are computational puzzles, we can use the same model to compute the hardness of productive puzzles. However, this hardness model do not give us the value of k — the number of tasks in a productive puzzle.

Also recall that the unit of the puzzle hardness d_{req} is given in number of instructions. Thus, for productive puzzles, the follow equality holds:

$$\begin{aligned} d_{req} =& d(T_1) + d(T_2) + ... + d(T_k) \\ =& \sum_{i=1}^{k} d(T_i) \\ \approx& k \cdot d_{avg} \end{aligned} \tag{5.21}$$

where, $d(T_i)$ is the number of instructions required to compute task T_i, and d_{avg} is the average number of instructions to compute any task. Solving this equation for k, we get

$$\begin{aligned} k =& \frac{d_{req}}{d_{avg}} \\ =& \frac{N(t)f}{\rho^* \mu} \frac{t_{req}}{t_{avg}} \frac{1}{d_{avg}} AF(t). \end{aligned} \tag{5.22}$$

5.2.4 Number of Known & Unknown Tasks

Once the number of task k to include a productive puzzle is determined, the next step is to figure out the numbers of known and unknown tasks p and u out of k. Assuming a desired value for maximum per-task error $\sigma_{max}(p, u, r_b, m)$ is given and k is already determined using Equation (5.22), one may attempt to derive p and u from Equation (5.12). However, there are 4 variables in Equation (5.12), and we only know the value of the σ_{max} and the value of the sum of two variables p and u, i.e., $k = p + u$. In order to derive the values of p and u, the fraction of malicious clients r_b and the majority threshold m must be assigned constant values.

Recall that the maximum per-task error rate σ_{max} can be written as a function of ε_{max}, i.e.,

$$\sigma_{max}(\varepsilon, m) = \sum_{d=m}^{2m-1} \binom{2m-1}{d} \varepsilon_{max}^{d} (1 - \varepsilon_{max})^{2m-1-d} \tag{5.23}$$

We can derive that σ_{max} is bounded by the following inequalities:

$$\frac{\varepsilon^m - \varepsilon^{2m}}{1-\varepsilon} - m\varepsilon^{2m-1} \leq \sigma_{max} \leq \frac{2^{2m-1}}{\sqrt{\pi m}} \frac{\varepsilon^m - \varepsilon^{2m}}{1-\varepsilon} - m\varepsilon^{2m-1} \tag{5.24}$$

The specific steps of deriving this inequality is given in the Appendix. As illustrated in Figure 5.7, Equation (5.24) provides a fairly tight upper and lower bound on σ_{max}.

Equation (5.24) gives us an idea of what ε_{max} should be for a given σ_{max}, but it still doesn't give us a formula to accurately compute ε_{max} from the given σ_{max}. Even with a fairly accurate formula to get ε_{max}, another formula is needed to derive p and u values. Equation (5.10) for $\varepsilon(p, u, w, r_b)$ has four variables, and even when the value of r_b is given, it is unclear how to get the value of w that maximizes ε.

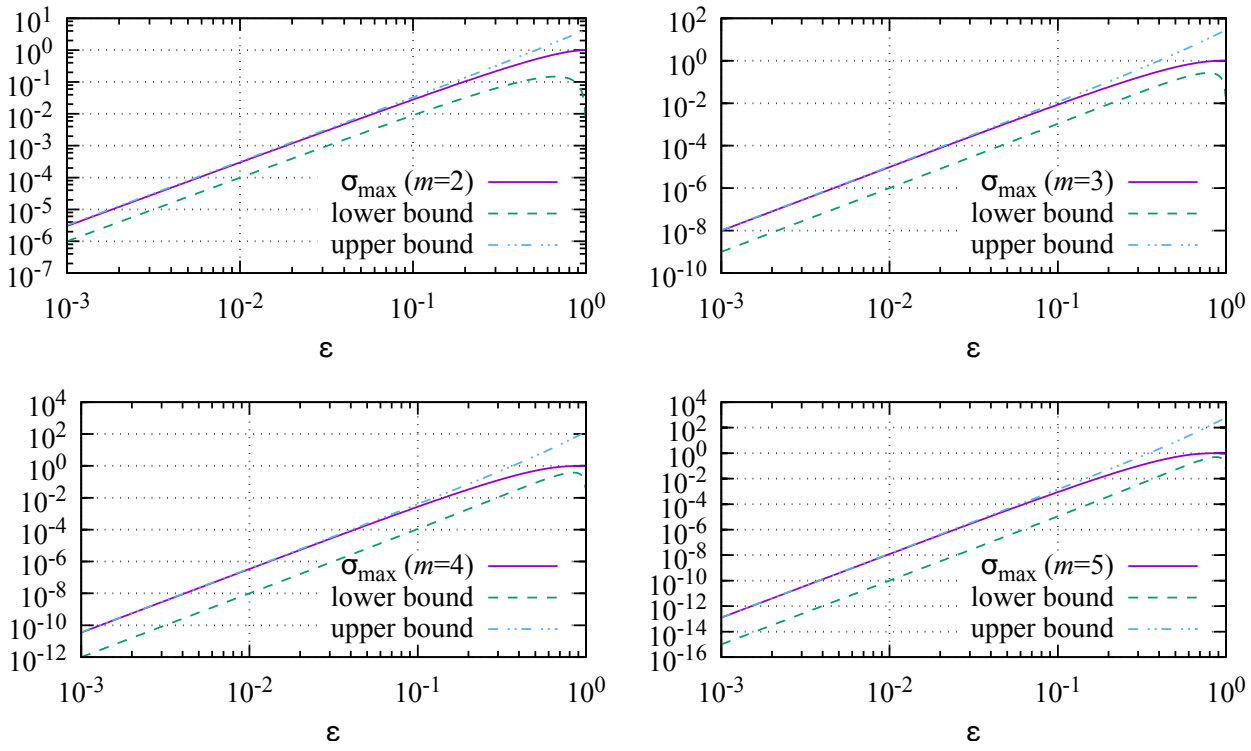


Figure 5.7: Lower and upper bound of maximum per-task error σ_{max}

Instead, we take a more practical approach to deciding the values of p and u. Figure 5.8 plots $\sigma(p, u, w, m, r_b)$ when u, w are varied and $m = 3$, $r_b = 0.1$, and $p = 2, 3, 4$. Regardless of what values u and w take, setting $p = 4$ can achieve a σ that never exceeds 10^{-3}. In our experiments, we set the number of known tasks $p = 4$, and derive the value of u by $u = k - p$ after determining k using Equation (5.22). Our experiment results show that this approach is viable and bounds the error below the expected maximum error.

A more accurate value of p can be derived for the special case of $w = k-1$ when a concrete value of the maximum per-solution error rate ε_{max} is given. Recall that $\varepsilon_{max} = \frac{r_b u}{(p+u)^2}$ is given in Equation (5.11) for the special case $w = k - 1$. The value of r_b cannot be known, but at maximum 100% of all clients are malicious, therefore r_b can safely be set to $r_b = 1$, which will give us an even tighter upper-bound on ε_{max}. By taking in $r_b = 1$ and $k = p + u$ into

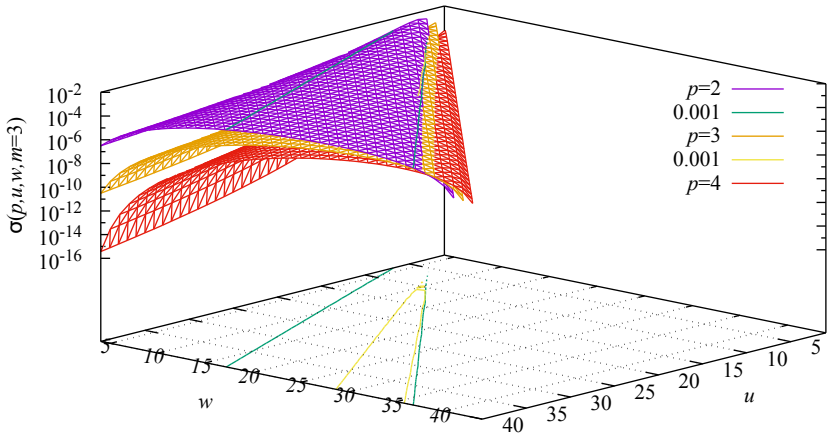


Figure 5.8: Max per-task error rate σ when $m = 3$, r_b=0.5

the ε_{max} equation, we get

$$\varepsilon_{max} = \frac{k-p}{k^2}. \tag{5.25}$$

Solving this equation for p, we get

$$p = k - k^2\varepsilon_{max} \tag{5.26}$$

Note that, the value of p becomes negative for some combination of k and ε_{max} values. To assure we always get a positive integer value for p, Equation (5.26) can be modified as follows.

$$p = max(0, \lceil k - k^2\varepsilon_{max} \rceil), \tag{5.27}$$

where, $max(x, y)$ is a maximum function that chooses the maximum of x and y, $\lceil x \rceil$ is the ceiling function. Once we determine p, we can easily get u using equation $u = k - p$.

5.3 EVALUATION OF DDOS DEFENSE EFFECTIVENESS

In this section, we evaluate the DDoS defense effectiveness of productive puzzles. A detailed description of the experimentation environment is provided, followed by the results corresponding to the experiments conducted.

5.3.1 Experiment Setup

The experiment setup for this evaluation assumes the evaluation framework described in Section 3.3 of Chapter 3, with the following additional configurations:

- An implementation of the Productive Puzzle Framework is provided within the Network Simulator 2 (NS-2) environment, although the implementation can largely be used outside of the NS-2 environment without significant modifications.
- The Task Server maintains the unknown tasks in batches, and a batch size of 100 is used. It waits for all tasks in the same batch to be completed before marking the batch as complete.
- To minimize the chances of a known task being re-used in a puzzle for the same client, the Task Server is configured to re-use the same known task at most P_{max} times. P_{max} is set to 5 in all of our experiments.
- Our experiments assume an unlimited supply of known and unknown tasks, and do not try to use bogus tasks, nor do they revert to using hash reversal tasks. This will allow us to measure the *cost of productiveness* more accurately.
- The *cost of productiveness* is measured in terms of how many known tasks are needed to solve a single unknown task. It can be approximately computed as follows.

$$c_p(P, U) = \frac{P_{total}}{U_{total}}, \tag{5.28}$$

where, P_{total} is the total number of known tasks used for computing a total number of unknown tasks U_{total}.

- The malicious clients are implemented to skip s many tasks in each puzzle, where $s = k - w$ and can be configured to vary between 0 (solving all tasks) and k (skipping all tasks). The malicious clients are also implemented to collude, in that they submit the same fake answers to the skipped tasks.
- All of our experiments use a *disparity factor* of 50, meaning the CPU frequencies of the fastest and slowest client machines have a ratio of 50 : 1. The client machine CPU frequencies are taken from a normal distribution.
- All malicious clients are configured to send requests at a rate that is 10 times the rate of the legitimate clients.

In addition to the three evaluation metrics — the legitimate utilization of the server, the average drop rate of legitimate requests, and the average end-to-end latency of legitimate requests — that are used in the experimental evaluations in previous chapters, we also measure the cost of productiveness c_p and per-task error rate σ.

5.3.2 Results

The first set of results are shown in Figure 5.9. They are obtained by varying the number of tasks to skip $(k - w)$ from 0 to 10. The number of known tasks p in each productive puzzle is set to $p = 4$, and the redundancy m is set to $m = 2, 3, 4$ respectively. The percentage of malicious clients r_b is fixed at 50%.

Results for the server utilization, request drop rate, and request latency metrics are plotted in Figures 5.9a, 5.9b, 5.9c, and 5.9d. First of all, changing the redundancy value m does not have an effect on these metrics, as expected. This is because redundancy is not a factor in deciding whether to grant a client service based on the solution it submits. The decision for granting service to a client request is based solely on the result of the known-unknown test, and redundancy does not play a factor in it.

On the other hand, the number of tasks to skip $(k - w)$ does have an effect on the normalized legitimate utilization of the server as well as on the average end-to-end latency of legitimate requests. The normalized legitimate utilization $\tilde{\rho}_{legit}$, which is computed by $\tilde{\rho}_{legit} = \frac{\rho_{legit}}{\rho}$, drops slightly as the number of skipped tasks $(k - w)$ increases. The average

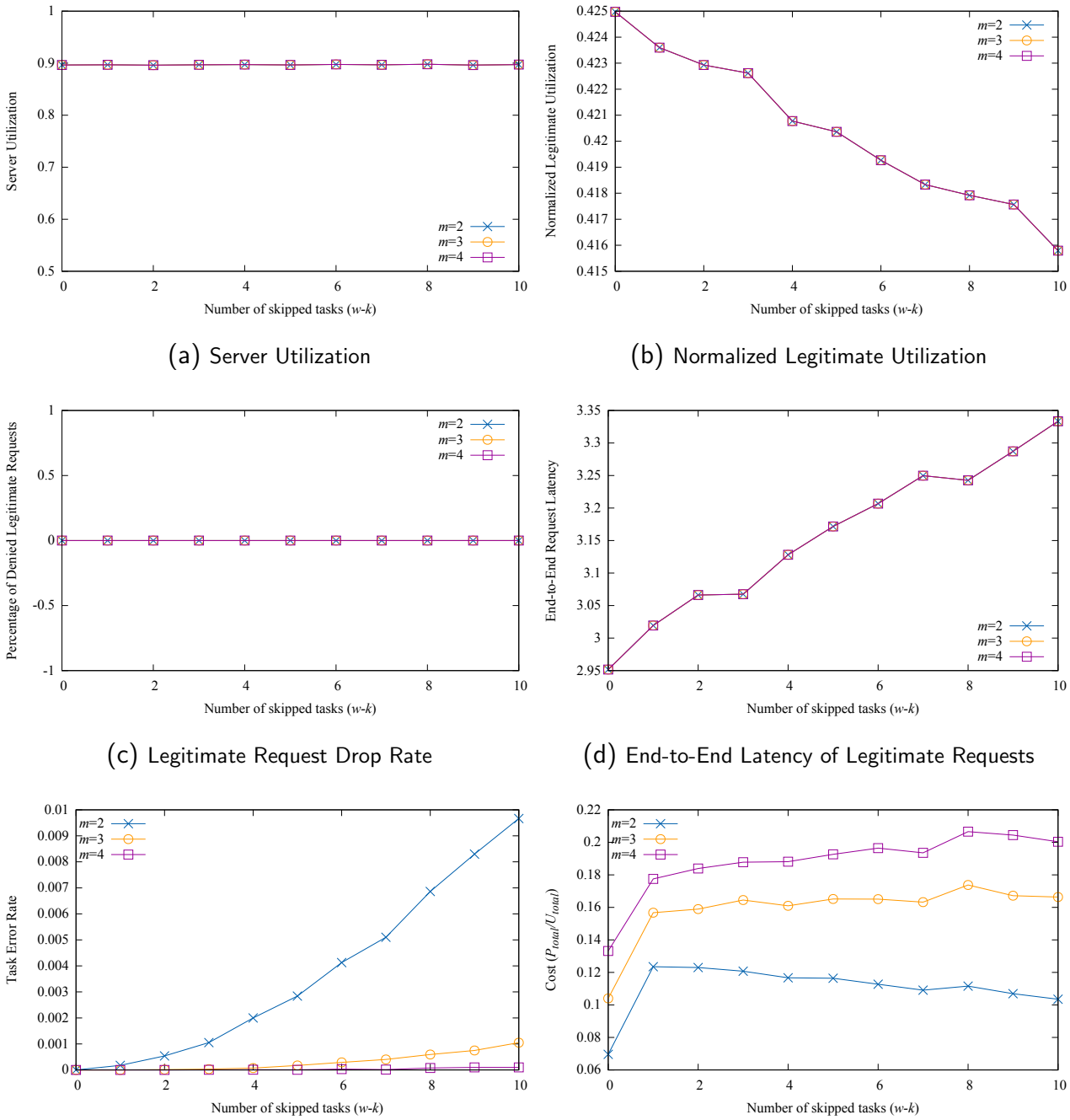


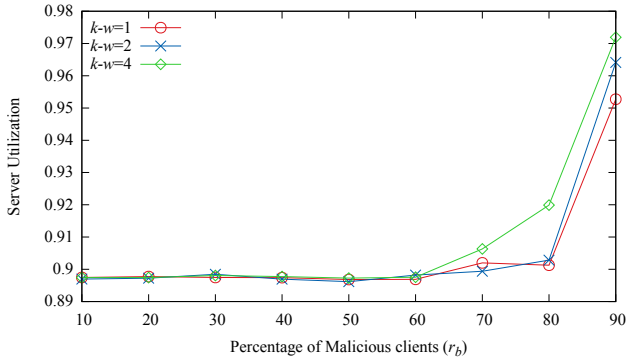
Figure 5.9: Effectiveness of Productive Puzzles DDoS defense for different configurations of number of tasks to skip $(w - k)$.

end-to-end latency of legitimate requests increases slightly as the number of skipped tasks increases. Both of the trends are attributed to the same factor: assuming the probability of passing the test is approximately constant, the malicious clients end up doing less work to complete a productive puzzle as $(k - w)$ increases, which leads to gains in the utilization of the server's capacity by the malicious clients. As the malicious clients return faster with solutions that passes the test for the same puzzle, it pushes the server utilization above the target utilization ρ^*, which in turn leads to more difficult puzzles, which ultimately leads to the larger request latencies seen in Figure 5.9d. The increase in the average end-to-end latency of legitimate requests and the slight decrease in the normalized legitimate utilization can be offset by adopting a strategy where clients that submit invalid puzzle solutions are blacklisted for a certain period of time. Such a strategy will only affect the malicious clients as legitimate clients are assumed to honestly complete the puzzles assigned to them.

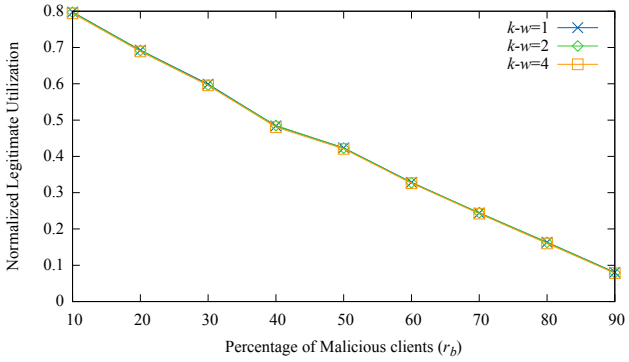
The per-task error rate σ increases as the redundancy m decreases, as shown in Figure 5.9e. This is expected behavior based on the probability of per-task error given in Equation (5.9). The per-task error rate σ also increases as the number of skipped tasks $k-w$ increases, which is also the expected behavior according to Equation (5.10). But, such increase is expected to be reach a maximum point then becomes monotonic decreases according to Equation (5.10). A more important observation is that when $m = 4$, the increase in per-task error rate is negligible. As such, the effect of the number of skipped tasks $k - w$ can be offset by adopting a larger value for redundancy m.

The cost of productiveness c_p is plotted in Figure 5.9f. The number of tasks to skip $k-w$ does not have an effect on the cost, as the cost stays approximately the same when $k - w$ is increased. On the other hand, the cost c_p increases as the redundancy m increases. This is expected since more known tasks needed to accompany the increased number of replicas of the same set of unknown tasks.

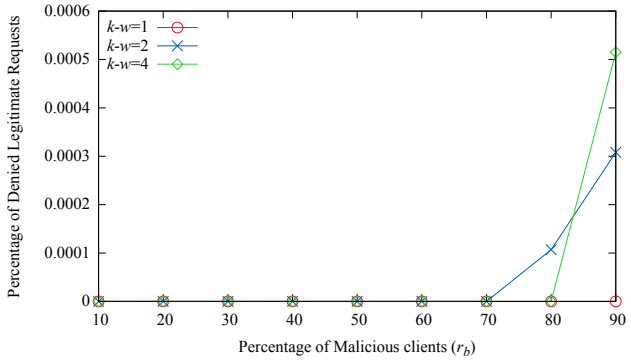
The second set of results are given in Figure 5.10. They are obtained by varying the attacker ratio r_b for the redundancy value $m = 3$ and the number of known tasks p fixed at $p = 4$. The number of skipped tasks $k - w$ is set to $k - w = 1$, 2, 4, respectively. The server utilization ρ is maintained around the target utilization ρ^* for almost all values of r_b, as shown in Figure 5.10a. However, starting from the point where $r_b = 70\%$, the utilization



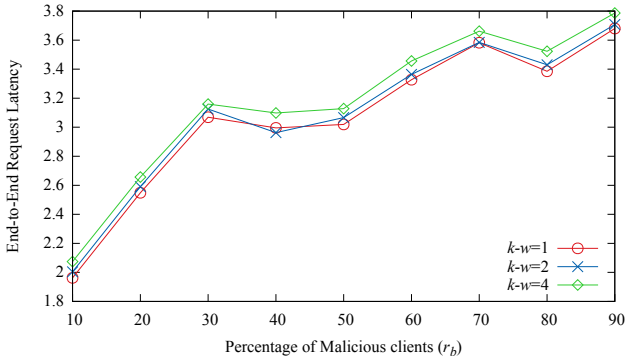
(a) Server Utilization



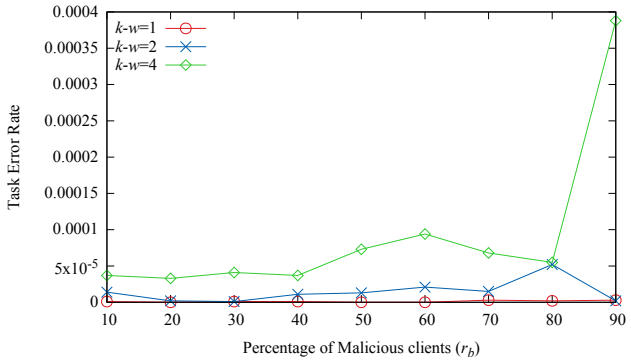
(b) Normalized Legitimate Utilization



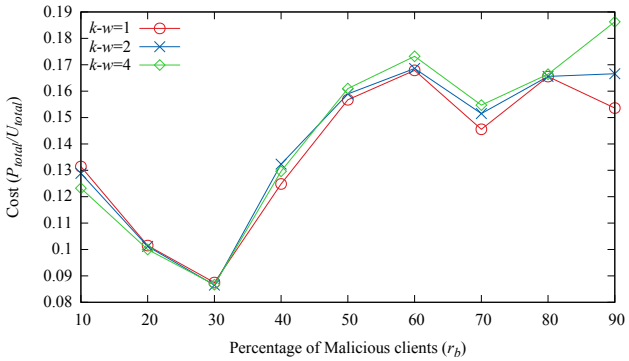
(c) Legitimate Request Drop Rate



(d) End-to-End Latency of Legitimate Requests



(e) Per-Task Error Rate (σ)



(f) Cost of Productiveness c_p

Figure 5.10: Effectiveness of Productive Puzzles DDoS defense under varying attack intensity.

of the server goes above the target utilization. The reason for this deviation from target utilization is that the cost of verifying the integrity of the puzzles become very significant as so many malicious clients are sending puzzle solutions to the server at a high rate. This is supported by the fact that the normalized legitimate utilization $\tilde{\rho}_{legit}$ stays proportional to the ratio of legitimate clients in the system, as shown in Figure 5.10b.

The percentage of dropped legitimate requests stays at zero for most values of r_b, and increases slightly after $r_b = 70\%$. We believe this is due to the same factor that the cost of verifying the puzzle integrity becomes very significant as the malicious clients become a supermajority, thus taking up a significant portion of the server's capacity which otherwise could've been partially used towards processing the legitimate requests.

The average end-to-end latency of legitimate requests increases as the percentage of malicious clients r_b increases, as shown in Figure 5.10d. This behavior is expected, since the dynamic adjustment of puzzle hardness, given in Equation (5.22), increases the puzzle hardness because of the increase of the server utilization beyond the target utilization rate ρ^*.

The per-task error stays mostly stable across different values of r_b, as shown in Figure 5.10e. On the other hand, the cost of productiveness c_p increases as the percentage of malicious clients r_b increases, as shown in Figure 5.10f. This result matches the expected result, since more known tasks will be wasted by the puzzle solutions that fail the known-unknown test as a lot more puzzle solutions that contain fake solutions to the skipped tasks are being submitted by the malicious clients. However, the takeaway from the results shown in Figure 5.10f and the results in Figure 5.9f is that the cost of productiveness is at an acceptable level, and can be further minimized by supplementing the productive tasks with hash reversal puzzle tasks.

5.4 CONCLUSION

Int this chapter, we proposed a novel puzzle idea called *Productive Puzzles* to transform the wasteful computations required by computational puzzles into computation of meaningful

tasks that provide utility. We introduced *known-unknown tests* to detect cheating clients with a very high probability and proved an upper-bound on the probability of successful cheating. To further minimize the chances of incorrect or bogus solutions being accepted, we incorporated majority voting based redundancy mechanism with the known-unknown tests. We showed that a very low error rate can be achieved using fairly small redundancy when majority voting is combined with the known-unknown test. Through extensive simulation study, we showed that the cost of computing the known tasks in productive puzzles is justifiable by the gain we get through completion of unknown tasks, as the gains are significantly larger. Our experiment results reinforced the per-task error bounds that we derived mathematically. While minimizing the wasteful work, productive puzzles maintained their effectiveness against DDoS attacks.

6.0 GUIDED TOUR PUZZLES

All computational puzzle frameworks require clients to perform expensive computational tasks, and results of such puzzle computations do not contribute towards solving meaningful problems that provide utility. We call this latter problem *wasteful computation problem*. Furthermore, computational puzzle schemes become less effective against a strong attacker that is equipped with powerful machines that are 10 or more times more powerful than almost all legitimate client machines, as shown in Section 4.6 of Chapter 4. We call this latter problem a *resource disparity problem*.

In this chapter, we introduce *guided tour puzzles*, a novel puzzle scheme that is not affected by the wasteful computation and the resource disparity problems. A guided tour puzzle requires a client to visit a predefined set of nodes, called *tour guides*, in a sequential order to complete a tour. Only after visiting all tour guides in correct visiting order can a client obtains a token that grants it service at the server. Guided tour puzzles not only achieve well-known desired properties of cryptographic puzzle schemes, but also better meet key requirements, such as *puzzle fairness* and *minimum wasteful computation*. The number of tour guides required by the scheme can be as few as two, and this extra cost can be amortized by sharing the same set of tour guides among multiple servers.

We evaluate the fairness of guided tour puzzles in a large-scale real network test-bed. We show that although variations exist in the tour delay for different clients, such variations are often significantly smaller than the possible variations in available computational power. More importantly, such variations cannot be manipulated by attackers towards their advantage to make their tour delay significantly smaller than all legitimate clients' tour delays. We also evaluate the effectiveness of guided tour puzzles in minimizing the impact of a denial of service attack in a realistic simulation environment using a large-scale network topology

that resembles the Internet. The simulation results show that the guided tour puzzle scheme provides an optimal defense against request flooding attacks and a near optimal defense against puzzle resisting attacks. Last, but not least, we look at the concurrent tours attack against guided tour puzzles, and provide an effectiveness defense against such attacks.

6.1 PUZZLE PROPERTIES AND DESIGN GOALS

In this section, we provide a comprehensive list of requirements that cryptographic puzzle schemes should satisfy, and explain the importance of each requirement.

Computation guarantee. The computation guarantee (also referred to as "bounds on cheating" [34]) means a cryptographic puzzle guarantees a lower and upper bound on the number of cryptographic operations spent by a client to find the puzzle answer. In other words, a malicious client should not be able to solve a puzzle by expending significantly less operations than required. This is discussed in [24].

Efficiency. The construction, distribution, and verification of a puzzle by the server should be efficient in terms of CPU, memory, bandwidth, hard disk, etc. Specifically, puzzle construction, distribution, and verification should add minimal overhead to the server to prevent the puzzle scheme itself from becoming an avenue for denying service [24] [4] [26].

Adjustability of hardness. This property is also referred to as *puzzle granularity* [69]. Adjustability of puzzle hardness means the cost of solving the puzzle can be increased at a fine granularity from zero to impossible [4]. Adjustability of hardness is important, because finer adjustability enables the server to achieve better trade-off between blocking attackers and the service degradation of legitimate clients.

Correlation-free. A puzzle is considered correlation-free if knowing the solutions to all previous puzzles seen by a client does not make solving a new puzzle any easier [4]. If a puzzle is not correlation-free, then it allows malicious clients to solve puzzles faster by correlating previous answers.

Stateless. A puzzle is said to be stateless if it requires constant memory at the server for storing client information or puzzle-related data. This property is discussed in [4].

Tamper-resistance. A puzzle scheme should limit replay attacks over time and space. Puzzle solutions should not be valid indefinitely and should not be usable by other clients [4] [26].

Non-parallelizability. Non-parallelizability means a puzzle solution cannot be computed in parallel using multiple machines [69]. Non-parallelizable puzzles can prevent attackers from distributing computation of a puzzle solution to a group of machines to obtain the solution quicker.

Puzzle fairness. Puzzle fairness means that a puzzle should take approximately the same amount of time to compute by any client, regardless of the CPU power, memory size, and bandwidth available to that client. If a puzzle can achieve fairness, then a powerful DoS attacker can effectively be reduced to a legitimate client. Not being able to achieve fairness leads to the resource disparity problem discussed earlier.

6.2 GUIDED TOUR PUZZLE

This section presents the GTP scheme. We start out with the main idea behind the GTP scheme, and describe a very basic puzzle protocol. Then, the limitations of the basic protocol is discussed and a solution is given to address each limitation.

6.2.1 The Basic Protocol

When a server suspects that it is under attack or its load is above a certain threshold, it asks all clients to solve a puzzle prior to receiving service. In the GTP protocol, the puzzle is simply a tour that needs to be completed by the client via taking round-trips to a set of special nodes, called *tour guides*, in a sequential order. We call this tour a *guided tour*, because the client should not know the order of the tour a priori, and each tour guide must direct the client towards the next tour guide. Each tour guide may appear zero or more times in a tour, and the term *stop* is used to represent a single appearance of a tour guide in a given tour.

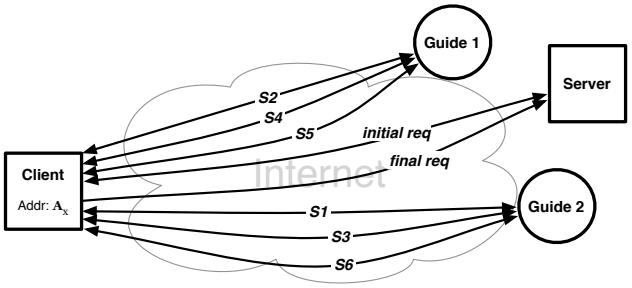


Figure 6.1: Example of a guided tour; the tour length is 6, and the order of visit is: $G_2 \rightarrow G_1 \rightarrow G_2 \rightarrow G_1 \rightarrow G_1 \rightarrow G_2$.

Figure 6.1 shows an example of a guided tour with two tour guides and 6 stops. The tour guide at the first stop of a tour is randomly selected by the server, and will also be the last stop tour guide, i.e., a guided tour is a closed-loop tour. The tour guide at each stop randomly selects the next stop tour guide. Starting from the first stop, the client contacts the tour guide at each stop and receives a reply. Each reply contains a token that proves to the next stop and the last stop that the client has visited this stop. Prior to sending its reply, the tour guide at each stop verifies that the client visited the previous stop tour guide, so that the client cannot contact multiple tour guides in parallel. After completing $L-1$ stops in a L-stop tour, the client submits the set of tokens it collected from all previous stops to the last stop tour guide (which is also the first stop tour guide), which will issue the client a proof of tour completion. The client then sends this proof to the server, along with its service request, and the server grants the client service if the proof is valid.

There are several issues concerning the basic protocol. First of all, a security mechanism must be in place to enforce the sequentiality of a single tour. Second, as a guided tour does not create a computational or bandwidth bottleneck at the client machine, an attacker may take many tours simultaneously, thereby qualifying itself for more resources of the server. Third, an attacker may cause DoS on the server indirectly by attacking the tour guides and the puzzle scheme itself. In the following subsections, we address each of these challenges individually.

Table 6.1: A summary of notations.

N	Number of tour guides in the system
G_j	j-th tour guide ($1 \leq j \leq N$)
k_S	Secret key only known to the server
k_{Sj}	Shared key between the server and G_j
$k_{i,j}$	Shared key between G_i and G_j ($i \neq j$)
L	Length of a guided tour
A_x	Address of client x
i_s	Index of the s-th stop tour guide ($1 \leq i_s \leq N$)
t_s	Coarse timestamp at the s-th stop of the tour
R_s	Client puzzle solving request at s-th stop
B	Size of the *hash* digest in bits

6.2.2 Ensuring Sequential Guided Tour

We set up N tour guides in the system, where $N \geq 1$. The server keeps a secret k_S that only it knows, and a set of keys $k_{S1}, k_{S2}, \ldots, k_{SN}$ are shared between the server and each tour guide. Each tour guide G_i maintains a pairwise shared key $k_{i,j}$ with every other tour guide G_j, where $i \neq j$ and $1 \leq i, j \leq N$. The total number of keys need to be maintained by each tour guide or the server is N, and this key management overhead is acceptable since N is usually a small number in the order of 10 or less. The tour length L is decided by the server to adjust the puzzle difficulty. Notations are summarized in Table 6.1.

The four steps of the GTP protocol is as follows.

6.2.2.1 Service request (I_1) A client x sends a service request to the server. If the server load is normal, the client's request is serviced as usual; if the server is overloaded, then it proceeds to the next step.

6.2.2.2 Initial puzzle generation (R_1)

The server replies to the client x with a message that informs the client to complete a guided tour. The reply message R_1 contains the following:

$$R_1 = \{L, i_1, t_0, h_0, m_0\}, \tag{6.1}$$

where i_1 is the uniform-randomly selected index of the first stop tour guide, t_0 is a coarse timestamp, h_0, m_0 are message authentication codes that provide message integrity. h_0 and m_0 are computed as follows:

$$h_0 = hmac(k_S,\ (A_x||L||i_1||t_0)) \tag{6.2}$$

$$m_0 = hmac(k_{Si_1},\ (A_x||L||i_1||t_0||h_0)) \tag{6.3}$$

where, $||$ denotes concatenation, A_x is the address (or any unique value) of the client x, and $hmac$ is a cryptographic hash-based message authentication code (HMAC) [36]. Since m_0 is computed using the key k_{Si_1} that is shared between the first stop tour guide G_{i_1} and the server, it enables G_{i_1} to do integrity checking later on.

6.2.2.3 Puzzle solving

After receiving the puzzle information, the client visits the tour guide G_{i_s} at each stop s, where $1 \leq s \leq L$, and receives a reply. Each reply message contains $\{h_s, m_s, i_{s+1}, t_s\}$, where i_{s+1} is the uniform-randomly selected index of the next stop tour guide, t_s is the timestamp at stop s, and h_s, m_s are computed as follows:

$$h_s = hmac(k_{i_s,i_1},\ (h_0||A_x||L||s||i_s||i_{s+1})) \tag{6.4}$$

$$m_s = hmac(k_{i_s,i_{s+1}},\ (m_{s-1}||A_x||L||s||i_s||i_{s+1}, t_s)) \tag{6.5}$$

At each stop s, the client sends a puzzle solving request message R_s that contains $\{h_0, L, s, t_{s-1}, m_{s-1}, i_1, i_s\}$ to the tour guide G_{i_s}, and the tour guide G_{i_s} replies to the client only if m_{s-1} is valid. In other words, each stop enforces that the client correctly completed the previous stop of the tour.

At the $(L-1)$-th stop, the tour guide $G_{i_{L-1}}$ knows that the next stop is the last stop, and replaces i_{s+1} with i_1 (recall that the first stop i_1 is also the last stop) when computing h_s and m_s. After completing the $(L-1)$-th stop, the client computes h_L as follows

$$h_L = h_1 \oplus h_2 \oplus \ldots \oplus h_{L-1} \tag{6.6}$$

where $\oplus$ means exclusive or, and submits $\{h_0, h_L, L, m_{L-1}, i_1, i_2, \ldots, i_L\}$ to the first stop tour guide G_{i_1}. Using these information, G_{i_1} can compute $h_1, h_2, \ldots, h_{L-1}$ using formula (6.4), and subsequently h_L using formula (6.6). Note that only G_{i_1} can compute values h_1 to h_{L-1}, since only it knows the keys k_{i_1,i_2} to $k_{i_1,i_{L-1}}$ that are used in the HMAC computations.

If the h_L submitted by the client matches the h_L computed by G_{i_1} itself, then G_{i_1} sends back the client a token h_{sol} that can prove to the server that the client did complete a tour of length L. The token h_{sol} is computed as follows:

$$h_{sol} = hmac(k_{Si_1}, (h_0||A_x||L||t_L)) \tag{6.7}$$

6.2.2.4 Puzzle verification The client submits to the server $\{h_0, h_{sol}, t_0, t_L, i_1\}$ along with its service request, and the server checks to see if h_0 and h_{sol} that it computes using formulas (6.2) and (6.7) matches the h_0 and h_{sol} submitted by the client. If both hash values match, the server allocates resources to process the client's request.

6.3 ANALYSIS

In this section we use analytical reasoning and experimental results to demonstrate that guided tour puzzles meet all of our proposed design goals.

6.3.1 General Puzzle Properties

For each property, we briefly explain how that property is achieved in guided tour puzzle.

Computation guarantee. Each client is required to complete L round-trips and perform L modulo operations that are necessary for finding which tour guides to contact. A client cannot skip any one of the required round-trips, because doing so leads to the wrong puzzle answer. Moreover, a client does not have an easier way to figure out which tour guide to contact next other than performing the inexpensive module operation. Therefore, guided tour puzzles achieve a strict computation guarantee that enforces the same number of operations for computing the same puzzle answer at all clients.

Efficiency. In guided tour puzzle, construction of a puzzle takes only a single hash operation to compute h_0 at the server, and verification of a puzzle answer takes one memory lookup in the improved scheme. Transferring of puzzle from server to the client requires $B/8$ bytes, where the size of hash digest B is usually $160 \sim 256$ bits.

Adjustability of hardness. The hardness of a tour puzzle is adjusted by adjusting the tour length L, which can be increased or decreased by one as needed. Therefore, guided tour puzzle provides linear adjustability of hardness.

Correlation-free. Guided tour puzzles are correlation-free, since knowing all previous puzzle answer does not help solve the current puzzle in any way. This property is provided by the security of the one-way hash chain we used in guided tour puzzle.

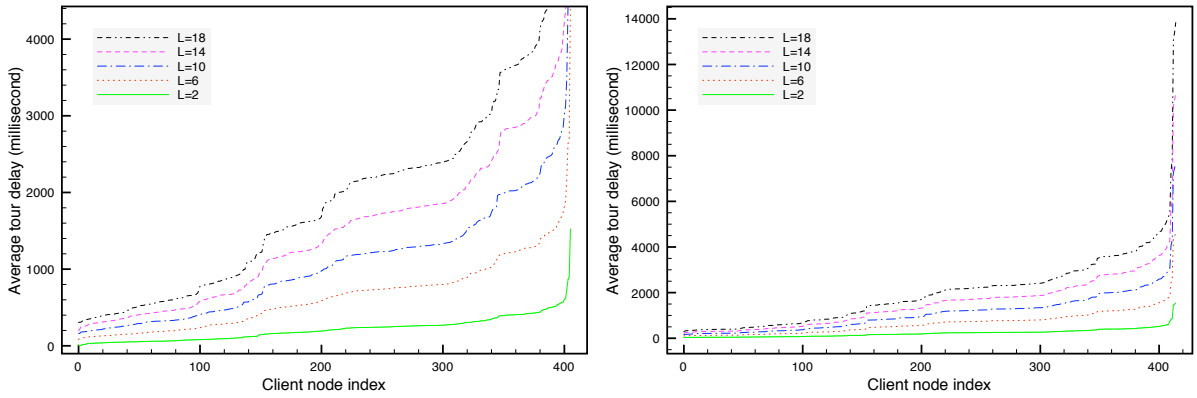
Stateless. Guided tour puzzle does not require the server to store any client or puzzle related information, except for the cryptographic keys that are used for the hash calculation. Puzzle answer verification using memory lookup does require few megabytes of memory. But the size of this memory is constant, and does not increase as the number of clients increase.

Tamper-resistance. The coarse timestamp used in the computation of each h_l guarantees a limited validity period of a puzzle answer. Meanwhile, the puzzle answer computed by one client cannot be used by any other client, since a value unique to each client is included in the computation of each h_l.

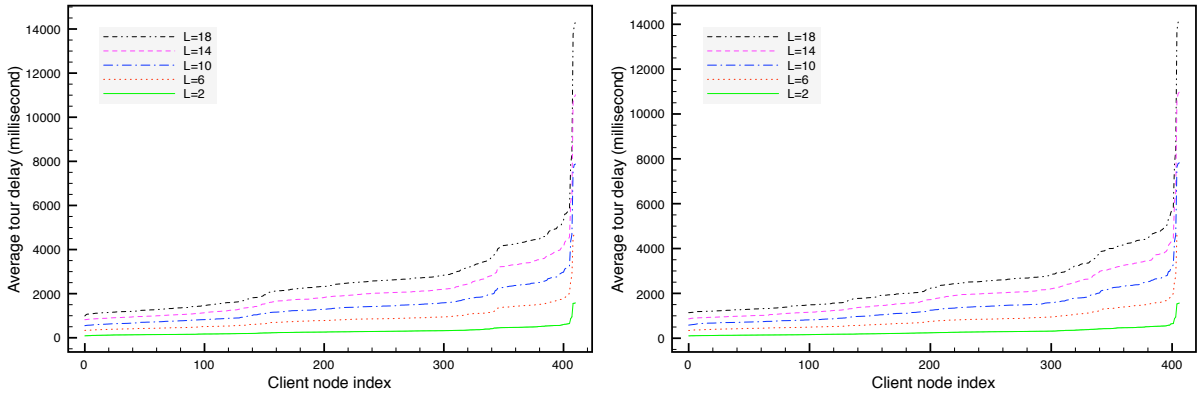
Non-parallelizability. Guided tour puzzle cannot be computed in parallel. An attacker with N malicious clients can assign each malicious client to contact one tour guide, and try to compute the puzzle answer in parallel. But each malicious client has to first get a h_l from the tour guide it is responsible for, and sends it to the next malicious client that is responsible for the next tour guide in the tour. Thus even with multiple malicious clients, attacker still has to compute the puzzle answer sequentially.

6.3.2 Achieving Puzzle Fairness

In the guided tour puzzle scheme, the time delay enforced on a client mainly comes from the round trip to multiple tour guides. The advantage of this is that not even a powerful attacker can control the round trip delay occurred in an Internet-scale distributed system.



(a) Tour delays at a single point in time, $N = 4$. (b) Average tour delays of two-week period, $N = 4$.

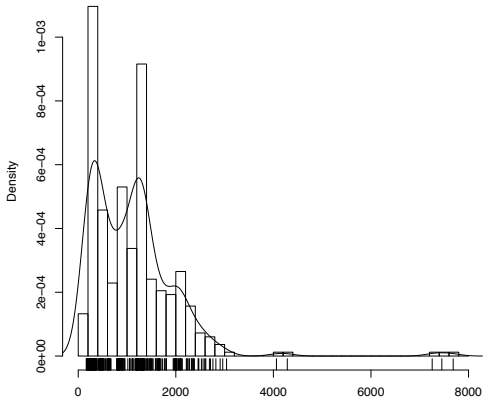


(c) Average tour delays of two-week period, $N = 8$ (d) Average tour delays of two-week period, $N = 12$

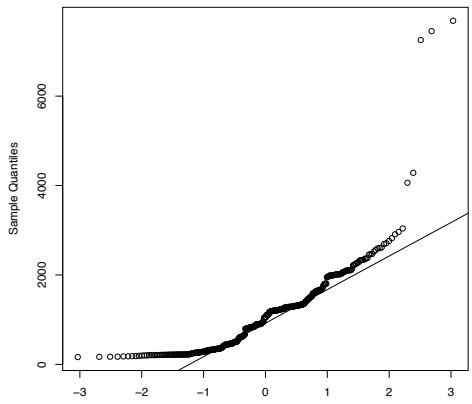
Figure 6.2: The tour delays of clients when different number of tour guides are used.

Due to the variation in the round trip delay across multiple clients, it is possible that the sum of round trip delays, which we will refer to as *tour delay*, varies across different clients. Next, we show that the variation in tour delay across multiple clients is within a small factor for a large distributed system such as the Internet. Note that this variation is different from the delay variation across multiple round trips for a fixed sender-receiver pair.

We used measurement data from PlanetLab Scalable Sensing Service (S^3) [65] that are collected over a two-week period. PlanetLab has a collection of over $1,000$ nodes distributed across the globe, and provides a realistic network testbed that experiences congestion, failures, and diverse link behaviors [37]. S^3 provides end-to-end latency data for all pairs of



(a) Probability density of tour delay when $N = 4$ and $L = 10$



(b) Tour delay Q-Q plot against normal distribution when $N = 4$, $L = 10$

Figure 6.3: Probability distribution of tour delays

nodes in PlanetLab. We use about 40% (over 400) of all PlanetLab nodes that have complete latency data throughout the two-week data collection period.

We randomly choose 20 nodes out of the previously selected 400 PlanetLab nodes as candidates for tour guides, and treated the remaining nodes as client nodes. The number of tour guides N is varied from 4 to 20, and the tour length L is varied from 2 to 18. For each (N, L) pair, we compute guided tours using formulas (6.2) and (6.4) for all client nodes, and compute a tour delay for each tour based on the collected data. As an example of tour delay at a single point in time, Figure 6.2(a) shows the tour delays of all client nodes for the setting $N = 4$ and $L = 4, 6, 8, 10, 12$ on May 23, 2009. For this particular data, the ratio of tour delays of the client with the most delay and the client with the least delay is 13, when the 5% clients nodes with abnormally large delays are excluded.

To give a better idea of how the tour delays vary across clients on average, we averaged tour delays of all clients over two-week period. To find the average tour delay of a client for a specific (N, L) setting, all tour delays of the client for that (N, L) during the two-week period are taken average. Then, the average tour delays are sorted from the least to the most, in order to provide a better view of delay variation across all clients. Figures 6.2(b), 6.2(c), and 6.2(d) show the average tour delays computed using this method for all client nodes

when $N = 4, 8, 12$. Results for other values of N are very similar to the results shown here. When excluding 5% client nodes with abnormally large delays, the ratio of tour delays of the client with the most delay and the client with the least delay is around 5. This disparity is several orders of magnitude smaller when compared to the disparity in available computational power (which can be in thousands [1] [23]). Figure 6.3(a) and 6.3(b) show that majority of tour delays are clustered within a tight area of delay and the distribution of tour delays closely simulates a normal distribution.

It is possible that a tour delay experienced by an attacker is significantly smaller than the delay experienced by a legitimate client for a single tour. However, the opposite case is also equally likely. Meanwhile, the variation in average tour delay across multiple clients is within a small factor as shown next by the experiment results. Although a small variation in the tour delay is inevitable, it cannot be effectively manipulated by an attacker to achieve unfair advantage over legitimate clients, regardless of attacker's CPU, memory, or bandwidth advantage. Therefore, guided tour puzzle achieves a fairness that is far better than any existing puzzle scheme can offer.

An attacker can try to minimize the puzzle solving time by using multiple malicious clients, where each malicious client is responsible for contacting the tour guide closest to it. But this kind of attacker actually cannot gain significant advantage over a legitimate client, because each malicious client has to wait one round-trip time to get the reply of the tour guide it is closest to, then spend another half a round trip delay to send this information to the malicious client closest to the next tour guide. Furthermore, the extra one-way delay is likely to be large, because the next tour guide is more likely to be far from the previous malicious client due to attacker's 'greedy' positioning of malicious clients.

6.3.3 Minimizing Wasteful Computation

In guided tour puzzle scheme, a client has to perform two types of operations: modulo operations for computing the index of the next tour guide, and sending packets to tour guides. To complete a guided tour puzzle with tour length L, a client only needs to perform L modulo operations plus send and receive a total of $2 \times L$ packets with about 20∼32 bytes

Table 6.2: The number of legitimate and malicious clients, and the load on the server.

% of malicious clients	0	10%	20%	30%	40%	50%	60%	70%	80%	90%
Malicious clients	0	190	380	570	760	950	1141	1331	1521	1711
Legitimate clients	1901	1711	1521	1331	1141	950	760	570	380	190
Offered load	0.96	1.82	2.69	3.55	4.42	5.28	6.14	7.0	7.87	8.74

(depending on the output size of the cryptographic hash function) of data payload. Since L is usually a small number below 30, this creates negligible CPU and bandwidth overhead even for small devices such as cellular phones. Therefore, we conclude that guided tour puzzles minimize wasteful computation on the client machines.

6.4 DDOS DEFENSE EFFICACY STUDY

We now evaluate the effectiveness of guided tour puzzle for preventing DDoS attacks. In this study, we focus our evaluation on the ability of guided tour puzzles in preventing the application layer DDoS attack. We show that the guided tour puzzle scheme provides an optimal defense against request flooding attacks and a near optimal defense against puzzle resisting attacks for the case where the server does not have the capability to differentiate the malicious clients from the legitimate ones.

6.4.1 Experiment Setup

The simulation framework setup and the valuation metrics are largely the same as the evaluation framework described in Section 3.3. Here, we only focus on the additional details or the differences in the configuration parameters.

The percentage of malicious client nodes is varied from 0% to 90% with an increment of

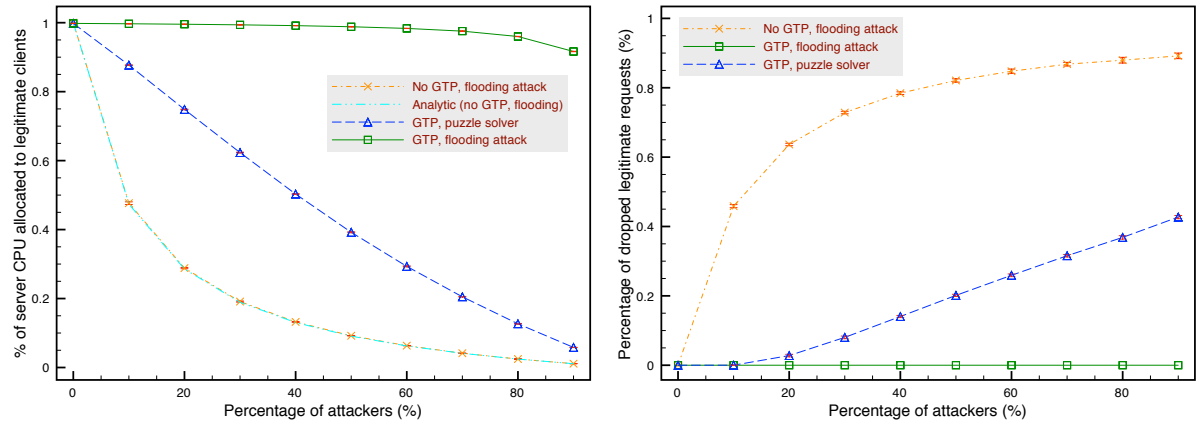
10%. Table 6.2 lists the server load and the number of malicious and legitimate nodes for different percentages of malicious clients. The server load is calculated as the ratio of the number of incoming requests per second to the server CPU capacity in requests per second.

Each client application is implemented as an ON/OFF source with ON/OFF period lengths are taken from a Pareto distribution with shape parameter α (also known as Pareto index) equals to 1.5 (NS-2 default). The average ON and OFF times are set to 2 seconds. Each legitimate client sends at an average rate of 8000 bits per second. The average client request size is set to 1000 bytes, thus each legitimate client essentially sends requests at one request per second during on times, and 0.5 request per second on average. The average ON and OFF times, the client request size, and the server response size values decided based on the Web workload model introduced by Barford and Crovella [7].

We experiment with two different types of attacks — the *flooding attack* and the *puzzle resisting attack*. In a flooding attack, a malicious client sends requests at a high rate and ignores the server's request for solving puzzles. In the puzzle resisting attack, a malicious client solves puzzles as fast as they can to send requests at the maximum speed possible. The latter is a much stronger attack, since a server that deploys guided tour puzzle scheme can trivially filter out a malicious request that contains incorrect puzzle solution, while a malicious request that includes correct puzzle solution consumes significantly more resources at the server.

The server capacity of $1,000$ requests per second is used so that the server's full capacity can be reached when all clients are legitimate, and the server load can be increased by 100% with each increase of the percentage of malicious clients (see Table 6.2). Using the average estimated client request rate of 0.5 request per second and the server CPU rate of $1,000$ requests per second, we can compute that the expected utilization of the server is $\frac{0.5\times1901}{1000}$ = 0.9505 when all the clients are legitimate clients. We achieved a utilization of 0.9656 for this setting in our experiments, which validates the correctness of our simulation setup.

To keep the simulation simple, instead of using an adaptable tour length, a fixed tour length is used within a single run of the simulation. For each solved puzzle, clients are granted service for a single request. We may achieve significantly better protection against the denial of service attack by dynamically adapting the tour length and the number of granted requests



(a) Percentage of CPU allocated to legitimate clients (b) Percentage of dropped legitimate requests

Figure 6.4: The effectiveness of guided tour puzzle against flooding attacks and puzzle resisting attacks (N=4, L=8).

per completed puzzle, but the simple scheme suffices for showing the effectiveness of guided tour puzzles.

The simulation length of each simulation run is set to 1000 seconds. For each simulation, a warmup period of 100 seconds is used; after which recording of the evaluation metric measurement is started. Each experiment is repeated 10 times using different random number generator seeds, and the average of 10 runs is reported along with a 99% confidence interval.

6.4.2 Results

The first set of simulations are conducted with a fixed tour length of 8 and using 4 tour guides. The results are reported in Figure 6.4 and 6.5.

6.4.2.1 Server CPU utilization Figure 6.4(a) illustrates the improvement in the percentage of the server's effective CPU capacity that is allocated to processing the requests of legitimate clients. As the line "No GTP, flooding" (GTP means guided tour puzzle) indicates, the legitimate clients' share of the server's CPU capacity drops rapidly as the percentage of attackers increases when no guided tour puzzle is used. The percentage of

server CPU allocated to processing legitimate requests in this case is predominantly decided by the ratio of total number of legitimate requests to the total number of requests. This can be validated by computing the percentage of legitimate requests for different percentage of malicious clients using the following formula.

$$\frac{r \times (1-x) \times N_c}{r \times (1-x) \times N_c \; + \; 10 \times r \times x \times N_c} = \frac{1-x}{1+9x} \tag{6.8}$$

where, r denotes the request rate of legitimate clients, N_c is the total number of client nodes, and x is the percentage of malicious nodes. The line "Analytic (no GTP, flooding)" is then computed using the Formula 6.8, and it overlaps perfectly with the experiment results from the NS-2 simulation for the case of "No GTP, flooding attack".

The top line "GTP, flooding attacker" in the Figure 6.4(a) shows that using guided tour puzzle eliminates the impact of flooding attackers entirely. In this scenario, the malicious clients do not solve any puzzle, but send requests that include fake puzzle solutions at a high rate in an attempt to consume as much server CPU capacity as possible. The slight decrease in the legitimate clients' utilization of the server CPU as the percentage of attackers increases is due to the increase in the percentage of server's CPU capacity allocated to verifying puzzle solutions. We intentionally used a low estimate of 10^6 hash operation per second as the server's hash computation rate to highlight the cost of puzzle solution verification.

The last line "Puzzle, solver" in Figure 6.4(a) is corresponding to the attack targeted at the guided tour puzzle scheme itself. It shows that the percentage of server CPU allocated to legitimate clients is roughly equal to the percentage of legitimate clients in the system when the guided tour puzzle scheme is used. We argue that without being able to differentiate legitimate clients from the malicious ones, the best a DoS prevention scheme can achieve is to treat every client equally and fairly allocate the server CPU to all the clients that are requesting service. Therefore, the optimal protection that a defense mechanism can provide without being able to differentiate malicious clients is to guarantee the legitimate clients the amount of server CPU that is equal to the percentage of legitimate clients in the system.

6.4.2.2 Request drops Figure 6.4(b) shows the percentage of dropped legitimate requests. When no guided tour puzzle is used, the flooding attack caused legitimate clients to

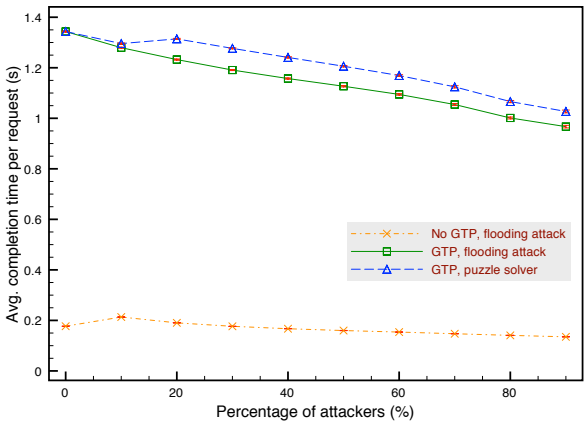
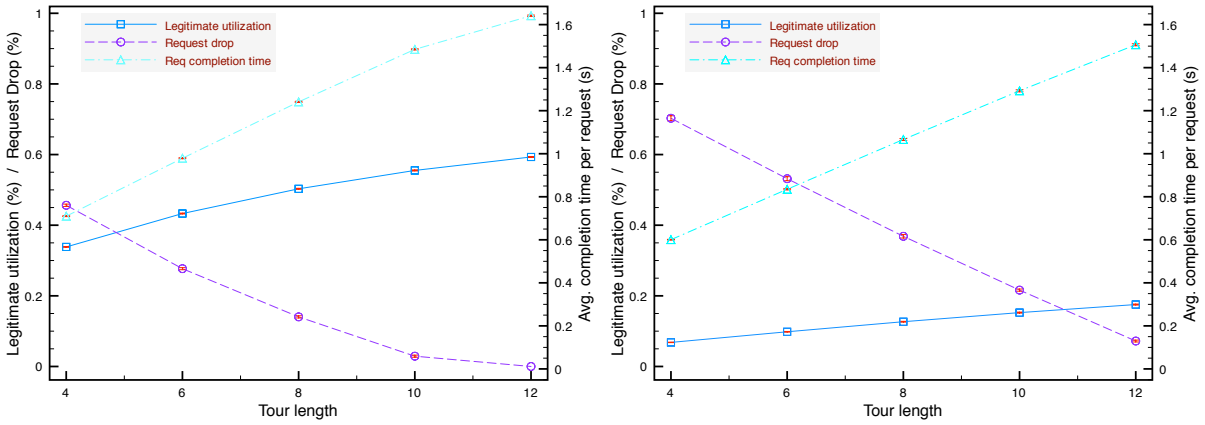


Figure 6.5: The cost of guided tour puzzles in terms of request completion times.

drop most of their requests as the line "No puzzle, flooding" indicates. When the percentage of attacker is increased to 90%, almost all legitimate requests are dropped as a result of the flooding attack. After switching to use guided tour puzzles (line "Puzzle, flooding"), the percentage of dropped requests becomes zero under the flooding attack even when the 90% of the clients are malicious. In the case of puzzle resisting attacks, guided tour puzzle scheme reduces the legitimate request drops by more than half in all cases, and reduces the request drops to zero in some cases. In fact, the legitimate request drops can be eliminated entirely even in the case of puzzle resisting attacks, as the simulation results in Section 6.4.2.4 show.

6.4.2.3 Request completion time Of course, the benefit of using the guided tour puzzle scheme comes at the cost increased average request completion time, as any other "proof of work" based DoS defense mechanism. This cost is shown in the Figure 6.5. When guided tour puzzle is utilized, the average completion time of a request increased significantly in both flooding attack and puzzle solver attack cases, due to the extra delay introduced by the puzzle solving process. Nonetheless, the increase in the request completion time is within an acceptable range of degradation of service quality. Moreover, the guided tour puzzle scheme provides an easy way to achieve a better trade-off between two mutually restricting sets of quality of service goals by varying the tour length.



(a) 40% attacker, N = 4 (b) 80% attacker, N = 4

Figure 6.6: The effect of the tour length on the effectiveness of the guided tour puzzle defense.

6.4.2.4 Effect of tour length The tour length in guided tour puzzles is critical for the optimality of the guided tour puzzle defense, especially for the legitimate clients' utilization of server CPU in the case of puzzle resisting attacks. The next set of simulation experiments are conducted to measure the effect of tour length on utilization, request completion time, and request drops in the case of puzzle resisting attacks. Configurations of 40% and 80% malicious clients are used in these experiments, and the number of tour guides N is set to 4.

The response of various metrics to the change in tour length is illustrated in Figure 6.6. As the tour length increases, the CPU allocated to legitimate clients ("Legitimate utilization") and the request completion time ("Req completion time") increase while the percentage of dropped legitimate requests ("Request drop") decreases. After increasing the tour length to 12, the percentage of dropped legitimate requests becomes zero, and the server CPU allocated to legitimate clients becomes optimal in both cases of 40% and 80% malicious clients. Here the optimal means legitimate clients are granted the amount of server CPU capacity that is equal to the percentage of legitimate clients in the system. Further increasing the tour length does not improve the utilization and request drop metrics and decreases the total utilization of the server CPU, while increasing the request completion time. The increase in the request completion time is evident since larger tour length means more round trips between clients

and tour guides. These observations tell us that choosing the right tour length is important in achieving optimal DoS prevention results and providing better trade-off between mutually restricting metrics.

6.4.2.5 Effect of the number of tour guides The last set of experiments are conducted to determine the effect of the number of tour guides on the effectiveness of guided tour puzzles. The 40% and 80% malicious clients are used, while the tour length L is set to 8. As the results in Figure 6.7 show, increasing the number of tour guides in the system does not produce any significant change in terms of all three metrics we are measuring. We can conclude from these results that guided tour puzzle can provide a good protection against the DDoS attack with just a few tour guides. Since the tour guides have a single function, which is replying to every request with the hash of the input message contained in the request, it is much easier to protect and maintain. The cost of hardware devices that can be used as tour guides likely to be significantly cheaper than over-provisioning by adding new servers. Moreover, a set of tour guides can be used to protect multiple servers, which further minimizes the cost per server by amortizing the total cost of tour guides over multiple servers.

6.4.3 Tour Guide Positioning

It is suspected that the positioning of tour guides in the network may have direct impact on the variation of the tour delay, affecting the puzzle fairness of guided tour puzzle. Thus a study of such possible impact is necessary. To measure the effect of tour guide positioning on the fairness achieved by guided tour puzzles, a large number of experiments are conducted with different tour guide positioning. Similar to the fairness evaluation in Section 6.3.2, round trip delay measurement data collected from the PlanetLab [37] Scalable Sensing Service [65] is used in all experiments.

We used 400 nodes that have complete delay data as client nodes. Another 20 nodes are randomly selected as candidate tour guides nodes, and 4 of them are chosen for each experiment. A total of $\binom{20}{4} = 4,845$ experiments are conducted to cover all possible combination

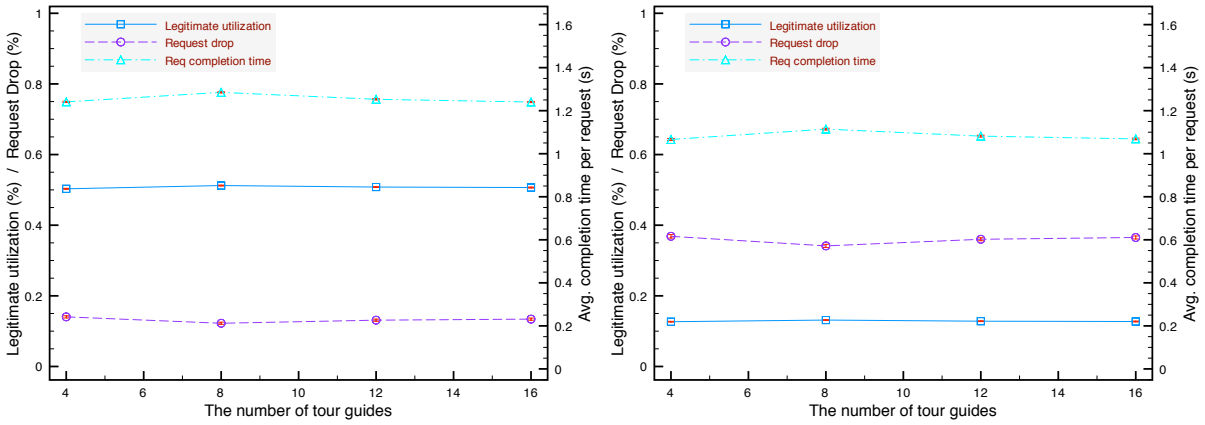


Figure 6.7: The effect of the number of tour guides on the effectiveness of the guided tour puzzle defense.

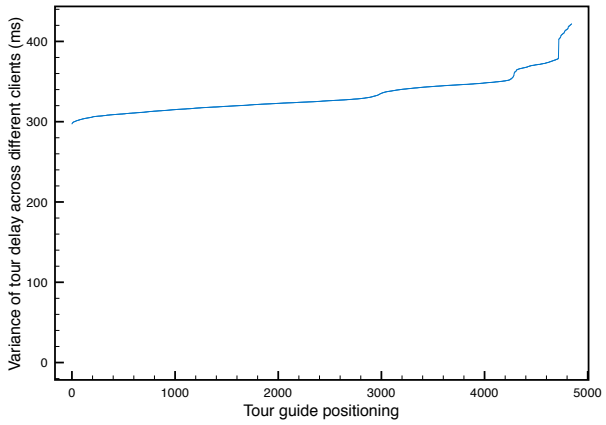


Figure 6.8: The effect of tour guide positions on the optimality of guided tour puzzle scheme fairness

of tour guide positioning. For each experiment, a tour delay variation across all 400 nodes is computed by calculating the standard deviation. The Figure 6.8 shows the plotted standard deviation (y axis) for all possible positioning of tour guides, for a total of 4,845 different positioning.

The experiment result suggests that positioning of tour guides does have certain effect

on the fairness achieved by guided tour puzzle. However, the impact seems relatively small as the results in Figure 6.8 show. As smaller the variation, the better fairness the guided tour puzzle scheme can provide, the way PlanetLab nodes are positioned can provide some insight on what might be a good positioning strategy. Based on the topology of PlanetLab nodes, it is concluded that the tour guides should be placed in the edge in the network as opposed to in the core, and each should be placed in the different domains as opposed to in the same autonomous domain. Further study is needed to provide more specific tour guide positioning guidelines for achieving better fairness, and we consider it to be a future extension of our work.

6.5 IMPROVEMENTS TO THE BASIC SCHEME

In this section, we provide a model for determining per-puzzle hardness. We also describe ways of improving the fault-tolerance and robustness of tour guides, and mechanisms to prevent replay attacks and concurrent tour attacks.

6.5.1 Determining Tour Length

Recall that our system model in Section 3.1 provides a simplified puzzle hardness model $\delta(t) = \frac{N(t)}{\lambda^*}$ for delay based puzzles in Equation (3.17), where $N(t)$ is the number of active clients in the system, and λ^* is the target arrival rate. We say this model is simplified because it does not take into account the cost of servicing a request and the current load of the server. Intuitively, requests that are more costly in terms of the server CPU time and amount of other resources spent should be given harder puzzles than the requests that are less costly.

Similar to the way we improve the puzzle hardness model for computational puzzles in Section 4.1 of Chapter 4, we improve Equation (3.17) and derive a per-puzzle hardness model

as follows.

$$d_{req} = \frac{N(t)}{\rho^* \mu} \frac{t_{req}}{t_{avg}} AF(t), \tag{6.9}$$

where, ρ^* is the target utilization of the server, μ is the service rate, t_{req} is the average time it takes the server to service request req, t_{avg} is the average time it takes to service any request, and $\frac{t_{req}}{t_{avg}}$ is the normalized cost of servicing the request req, $AF(t)$ is the Adjustment Factor that was given by Equation (4.3) and (4.4). In this equation, puzzle hardness d_{req} is given in terms of delay time in seconds, but we need the tour length. The tour length for a given request req can be derived by dividing Equation (6.9) by the average round-trip delay RTT_{avg} between the clients and tour guides, i.e.,

$$L_{req} = \frac{d_{req}}{RTT_{avg}} = \frac{N(t)}{\rho^* \mu RTT_{avg}} \frac{t_{req}}{t_{avg}} AF(t), \tag{6.10}$$

Note that CPU frequency is not a factor in this tour length model, which supports the observation that tour puzzles are not effected by the computational power of the clients. On the other hand, it is effected by the round-trip delay between clients and the tour guides. It is shown in Section 6.3.2 that the disparity in tour delays can be between 5-13, which is much smaller than the disparity in computational puzzle solving times. But more importantly, the tour delays are normal distributed and it is extremely for an attacker to have all its malicious clients get 10-15 times smaller tour delay than legitimate clients. We showed in Section 4.6 of Chapter 4 that puzzles can still be very effective if the disparity in a certain resource does not descriminate between the legitimate and malicious clients, meaning malicious clients do not consistently have multiple times the resources of legitimate clients. Tour puzzles fits this profile perfectly, and since all malicious clients cannot consistently have orders of magnitude smaller tour delay than all legitimate clients. Therefore, tour puzzles can still be very effective even when the tour delays have a disparity factor between 5-13 or more.

6.5.2 Increasing Tour Guide Robustness

To prevent attackers from indirectly launching DoS on the server by attacking one of the tour guides, tour guides should be robust against attacks on themselves. As tour guides perform

very simple operation, i.e., computing a hash function, they are very light-weight and far less susceptible to DoS attacks. Also due to their simple operation, securing the tour guides against compromise attempts also becomes much simpler. Furthermore, the basic guided tour puzzle scheme is designed to localize the impact of a compromised tour guide. Due to the all-pair pair-wise shared keys, compromising one tour guide only gives the attacker a free ride for the leg of the tour that starts with the compromised tour guide, and the attacker still has to complete the majority of the tour.

Although the tour guides are highly immune to DoS attacks, it is still possible for a tour guide to be down due to internal failure or a very strong DoS attack that involves millions of nodes. To operate gracefully when one of the tour guides is down, all tour guides exchange heartbeat messages with each other and with the server, such that unavailability of a tour guide is immediately known by the server and other tour guides, and forwarding of clients to the failing or unavailable tour guide is avoided. The heartbeat messages should be adequately protected to thwart attacks on the tour guides.

6.5.3 Preventing Replay Attacks

Puzzle solution replay attacks can be a big problem for Guided Tour Puzzles as well if not prevented. We proposed Auto-Expire Cache based replay attack prevention in Section 4.2 of Chapter 4, and showed that the solution is very effective in preventing replay attacks in Section 4.5. As the same solution can be used without any modification in any puzzle scheme, we propose to adopt the same replay prevention solution for Guided Tour Puzzles.

6.5.4 Preventing Concurrent Tours

The design of Guided Tour Puzzle in Section 6.2 considers the non-parallelizability of a single tour puzzle, but does not fully consider non-parallelizability of multiple puzzles to be solved by a single client. Without a mechanism in place to prevent it, a malicicous client can issue a separate request to get another tour puzzle while it is still in the process of completing an already started tour. To make matters worse, a malicious clients can issue many tour requests one after the other and carry out multiple tours concurrently. We call the DDoS

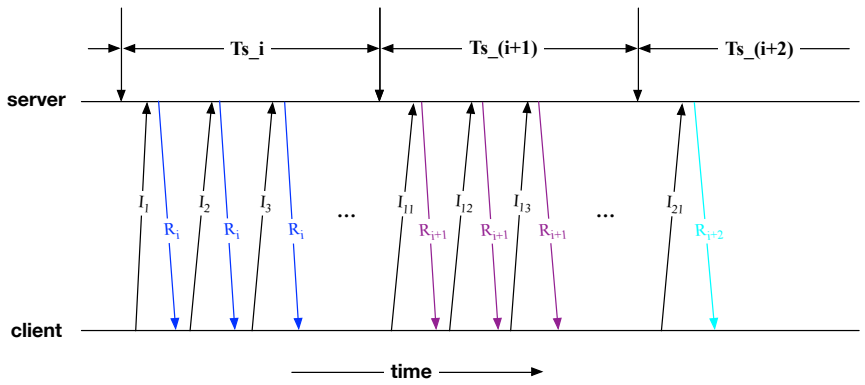


Figure 6.9: Response to concurrent tour requests in the same time period

attacks that utilize such concurrent tours *concurrent tour attacks*.

One possible defense against the concurrent tour attack is to allocate the usage duration of resource rather than allocating the resource itself. Specifically, each puzzle solution grants the client access to the server for a time duration T_s. The time unit for the corase timestamp that is used in the puzzle generation and verification will also be set to T_s. What this implies is that all requests by the same client within the same time duration T_s will get the same puzzle that grants service for the same time period. With such configuration, it does not make sense for a client to request multiple puzzles and do multiple tours because all of them will result in the same token that grants service for the same time period. This concept is illustrated in Figure 6.9.

The "same puzzle" effect can be achieved without significant modification to the puzzle generation algorithm of Guided Tour Puzzles described in Section 6.2. Recall that the server's puzzle reply message $R_1 = \{L, i_1, t_0, h_0, m_0\}$ contains a coarse timestamp t_0. The server can simply use T_s as the smallest unit of this coarse timestamp, which means within the same time period, this coarse timestamp will always be the same in all puzzle reply messages. Then, the server can check this coarse timestamp that will also be included in the puzzle solution to determine for which time period the solution is for. A simple server strategy can only accept the solution if it is for the current time period, and reject all solutions for

any past time periods. This strategy could be problematic when the client's puzzle request arrives towards the end of the current time period; but if the time period length T_s is set to a big enough value, such cases will occur rarely; and even when they do occur, retrying the entire puzzle will give the client plenty of time to return with a solution in the next time period.

The caveat of this "one puzzle per time period" based solution is that further per client based enforcements are needed to prevent malicious clients from flooding the server immediately after acquiring access during each time periods. We call such "flooding after acquiring access" type of attacks *pulsing attacks* to differentiate them from the continuous flooding attack. One can notice the similarity between the pulsing attack and the replay attacks: both acquires a valid "token" for service and floods the server with requests after that, until some expiration time arrives; and repeats the whole process for the next puzzle or time period. Such similarities lead us to the intuition that the same solution may apply to both.

The Auto-Expire Cache based solution to the replay attacks does indeed apply to this pulsing attack, can be used as following to counter the pulsing attack:

- Once a tour puzzle solution is accepted by the server, it adds the corresponding service token into the Auto-Expire Cache, along with a token bucket [32] for regulating the requests associated with that token; Note that there is only a single token per client per time period, and completing multiple tours in per time period will lead to the same token for the same client. Meanwhile, token bucket is not the only algorithm can be used here, other efficient mechanisms or algorithms for rate limiting requests can be used as well.
- For every request that is associate with the same token, the server applies the corresponding rate limiting algorithm before accepting the request into service queue.
- The duration of the Auto-Expire Cache is set to the length of a single time period T_s. Therefore, at the end of each time period, the Auto-Expire Cache automatically removes the items associated with that period, thus limiting the memory for keeping the state information.

This combination of one-puzzle per time period mechanism and rate limiting of admitted tokens stored in Auto-Expire Cache works well, but somewhat complicated to implement.

Moreover, it cannot take advantage of the regulation of request inter-arrival times by the tour puzzles, and requires a separate rate limiting enforcement on a per-client basis.

Instead, we adopt a simpler *concurrent tour detection* approach, which also uses Auto-Expire Cache. The idea is to keep the unique identifier of the client in the Auto-Expire Cache every time the client is assigned a puzzle, and keep the client ID for a duration of t_{DI}, where t_{DI} is called the *concurrent puzzle detection interval*. Meanwhile, every time the client requests for puzzle, check this client ID cache and send a puzzle if the client's ID is not in the cache; ignore the client if its ID is already in the cache. Now, if a malicious client tries to perform concurrent tours, it will be detected in its puzzle requests are arriving less than t_{DI} time apart from each other. The malicious client can certainly space out its requests by at least t_{DI} to avoid the detection, but then it can no longer perform multiple tours concurrently by doing so.

This concurrent puzzle-solver detection mechanism can be combined with the already existing token cache that is used for preventing replay attacks to eliminate all false positives — the likelihood of labeling legitimate clients as concurrent puzzle solvers. A false positive happens when a legitimate client, whom just received service after completing a tour, wants to do another tour, but its client ID is still in the client ID cache used by the concurrent puzzle-solver detection mechanism. Such false negatives can be avoided by synchronizing the client ID cache for concurrent puzzle solver detection and the token cache used for the replay attack prevention. By 'synchronization', we mean that both caches have the same cache entry expiration interval value Δ, and starts their intervals at the same time. With such configuration in place, we first check that if a client has a token in the token cache when deciding whether to allow a tour for a client. If the client has a token in the token cache, it mean completed the previous tour already so it is okay to grant it another tour; if the client does not have a token, then we check the client ID cache to decide whether to grant client a new tour.

The concurrent puzzle-solver detection mechanism is described more formally in Algorithm 6.1.

Algorithm 6.1: Concurrent Puzzle-Solver Detection

Data: clientIdCache - the Auto Expire Cache of client IDs for concurrent puzzle detection;
tokenCache - the Auto Expire Cache of tokens used in replay attack prevention;
Δ - the cache entry expiration interval;
current_time - the timestamp of current time.

```
1  clientIdCache.setExpirationInterval(Δ)
2  tokenCache.setExpirationInterval(Δ)
3  function isEligibleForPuzzle(clientId)
4      if tokenCache.contains(clientId, current_time) or not clientIdCache.contains(clientId,
       current_time) then
5          return true
6      else
7          return false
```

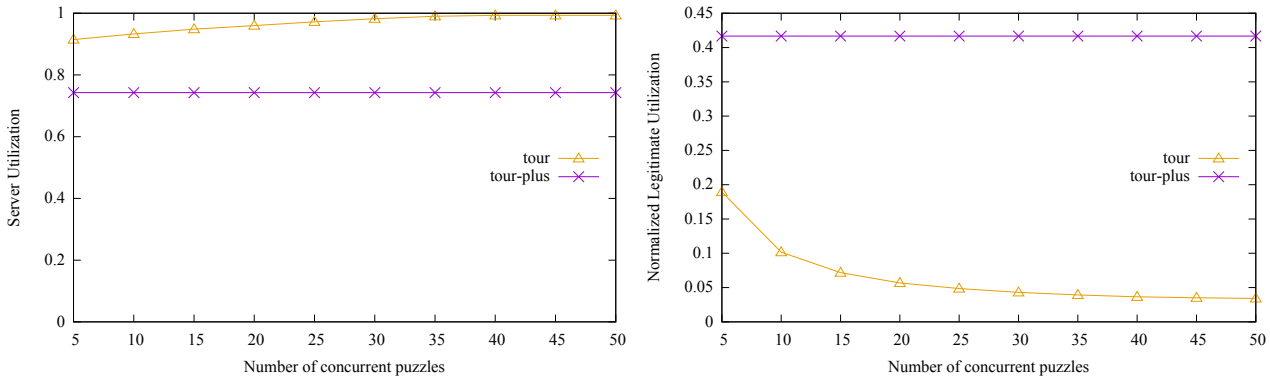
6.6 EVALUATION OF CONCURRENT PUZZLE SOLVING DEFENSE

In this section, we evaluate the guided tour puzzles with the improvements proposed in the previous section. Specifically, we incorporate the dynamic tour length determination, replay attack prevention, and concurrent puzzle prevention techniques introduced. Since the advantages of dynamic per-request puzzle hardness determination and Auto Expire Cache based replay attack prevention are already being demonstrated in Section 4.5 of Chapter 4, we only focus on the evaluation of the defense against the concurrent tours attack.

6.6.1 Experiment Setup

The setup of the experimentation environment is the same as the one described in Section 6.4.1, with a few changes and additions that are given by the list as follows.

- We use a smaller network topology that consists of a total 342 nodes with 236 client nodes, so that we will be able to run enough simulations with different settings in a reasonable amount of physical time. As the simulation of concurrent tours attack increases the number of simulation events generated 5 to 50 times of the numbers without the concurrent tours, it becomes impossible to complete a single run of simulation in a reasonable amount of time using the original 5000 node topology. A comparison of



(a) server utilization (b) normalized legitimate utilization

Figure 6.10: Utilization during the concurrent tours attack

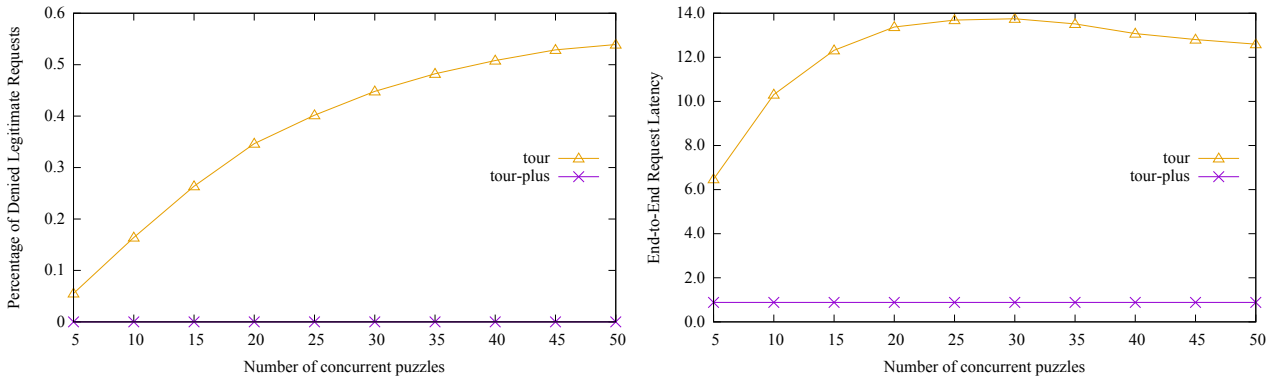
several experiments results with this smaller topology to the results that use the 5000 node topology shows that both provide us with the same knowledge about the system under study.

- We fix the ratio of malicious clients to 50% for all experiments.
- A malicious client that performs concurrent tours is implemented by modifying the tour puzzle resisting attacker to perform m tours concurrently, where m is varied between 5 to 50 with an increment of 5.
- The cache entry expiration interval for both the client ID cache and the token cache are set to 2.5 seconds.

6.6.2 Results

The experiment results reported here compares the guided tour puzzle defense with and without the concurrent tours detection mechanism. The results are given in Figure 6.10 and 6.11. Lines labeled with 'tour' correspond to puzzle defense without the detection mechanism, and lines labeled with 'tour-plus' correspond to puzzle defense with the detection mechanism.

The utilization of the server during the concurrent tours attack is given in Figure 6.10a.



(a) Percentage of denied legitimate requests

(b) Request latency

Figure 6.11: Legitimate request drops and latency during the concurrent tours attack

Without the concurrent tours detection, the guided tour puzzle defense starts to loose control of the server utilization rate starting when each malicious client performs 20 tours in parallel. In contrast, the guided tour puzzle defense with the concurrent tours detection controls the server utilization well above the maximum server utilization of 0.9.

Figure 6.10b shows the normalized legitimate utilization. Recall that normalized legitimate utilization is obtained by dividing the legitimate utilization by the server's total utilization, i.e., $\tilde{\rho}_{legit} = \frac{\rho_{legit}}{\rho}$. It tells us how many percentage of the server's resources are spent on processing legitimate requests out of the total resources spent. As we can see that the guided tour puzzle defense performs poorly without blocking the concurrent tours, and the legitimate clients get far smaller share of the server time than their fair share. On the other hand, the guided tour puzzle defense performs superbly with the help of concurrent tours detection mechanism, and guarantees around 42% of the server's time for processing legitimate requests. This is very close to their fair share, since there are 50% legitimate clients in the system.

The average percentage of legitimate requests that are denied due to overflow of the service queue is given in Figure 6.11a. The percentage of denial legitimate requests gets over 50% when concurrent tours are allowed. For the other 50% of legitimate requests, the

average end-to-end request latency become unacceptably high, nearing as much as 14 seconds, as shown in Figure 6.11b. Note that the end-to-end request latency includes the tour delay, so majority of the high request latencies in this case are due to tour delays. When there is no detection and blocking of concurrent tours, the guided tour puzzle defense attempts to dynamically increase the tour length to counter high utilization of the server, but that could not sufficiently deter the malicious clients; and instead it only increases the latency of legitimate requests. When concurrent tours detection mechanism is used, it guarantees all legitimate requests service while also keeping the request latency to a minimum degree.

Overall trend can be observed from all experiment results reported in this section is that the guided tour puzzle defense performs equally well for all different configurations of concurrent tours when it adopts the concurrent tours detection mechanism, for example it gives the same results when the number of concurrent tours is 5 or 50. This demonstrates that the concurrent tours detection algorithm we proposed works very well without false positives, and makes the guided tour puzzles very robust against attacks that involve concurrent tours.

6.7 CONCLUSION

In this chapter, we explored the idea of delay based puzzles, motivated by solving the wasteful computation and resource disparity problems of computational puzzle based DDoS defense mechanisms, and proposed guided tour puzzles. We showed that it achieves the desired properties of an effective and efficient cryptographic puzzle scheme. In particular, we showed how guided tour puzzles achieve puzzle fairness while eliminating wasteful computation requirements of computational puzzles. By measuring the tour delays on an actual network test bed environment, we showed that the variation in the tour delays are smaller in comparison with the disparity in the computational powers. More importantly, we showed that tour delay variation cannot be effectively manipulated by malicious clients to achieve unfair advantage over legitimate clients. Meanwhile, using extensive simulation studies we showed that guided tour puzzle is very effective in mitigating distributed denial of service attacks, and that it is a practical solution to be adopted.

7.0 STOCHASTIC FAIR DROP FRAMEWORK

In previous chapters, we focused on devising effective and efficient puzzle mechanisms to defense against DDoS attacks. In this chapter, we shift gears and explore the possibility of achieving robust defense against DDoS attacks using active queue management and request dropping techniques. In particular, we introduce *Stochastic Fair Drop (SFD)* framework, and discuss the policies and mechanisms it adopts in order to achieve a very effective defense against DDoS attacks.

According to Definition 1.2 given in Chapter 1, the goal of a denial of service attack is to make the targeted service unavailable to its intended users for a time that significantly exceeds the intended waiting time. Often, this goal is achieved by sending overwhelmingly large amounts of requests to the targeted server. As a result, the *queue* used by the targeted server (whether that server being a router, a TCP server, or a Web server) will be mostly occupied by requests from the attacker(s).

Fair queuing approaches, such as [8, 20, 50, 54, 66], aims to prevent a set of users from taking up most of the resources by fairly allocating the server's resources among all of its users. However, such an approach becomes ineffective when attackers can spoof or mint at will the unique identifiers, such as IP address or flow ID, that the fair resource allocation mechanisms critically depend on. If a single malicious user can assume many different identities by spoofing or minting many unique identifiers, then a fair queuing mechanism unwittingly allocates the malicious user a share of the server resources that far exceeds its fair share. In fact the single malicious user will get a share that's proportional to the number of spoofed identities it assumes.

A key observation about the state of the server that is under denial of service attack is that the server's queue is filled mostly with the attacker's requests. So, if an item is chosen

randomly out of all items in the queue, the probability of the chosen item belonging to the attacker is significantly higher than the probability of that it belongs to a legitimate user.

With this insight regarding the queue state of the server under denial of service attack, we propose a simple request drop policy, called *Stochastic Fair Drop*. Under Stochastic Fair Drop policy, all requests in the queue have equal chances of being dropped on every request dropping decision. Since there are more bad requests during the attack than there are good ones, for each request drop decision, the probability of choosing a malicious request is significantly higher than choosing a legitimate request. This is true regardless of whether malicious requests are coming from a larger number of attack sources or a small set of attackers, since both cases will result in more malicious requests in the service queue than the legitimate ones. The attacker can decrease the probability of its requests being dropped only by actually decreasing its aggregated request rate.

To be able to efficiently enforce Stochastic Fair Drop policy, the service queue not only should provide efficient *enqueue* and *dequeue* operations, but it should also provide an efficient *random delete* operation. Yet, none of the existing data structures provide $O(1)$ constant time complexity for all three operations. To this end, we design a novel data structure, called *Indexed Linked List*, and show that it satisfies the stringent efficiency requirements for supporting Stochastic Fair Drop policy.

To make Stochastic Fair Drop policy even more effective against the DDoS attacks, we enhance it with a simple but very effective malicious user detection and blacklisting mechanism. This new detection mechanism utilizes the fact that malicious requests are more likely to be dropped under Stochastic Fair Drop policy, and blacklists a client if multiple requests from that client are being dropped within a given time period. This simple detection and blacklisting mechanism turned out to be very effectiveness in filtering out malicious and misbehaving clients without affecting the legitimate ones.

Our extensive experimental evaluation results show that, Stochastic Fair Drop framework provides a robust defense against DDoS attacks on both the application layer as well as the network layer of the Internet protocol stack.

7.1 STOCHASTIC FAIR DROP

In this section, first give a more in-depth description of Stochastic Fair Drop policy, followed by the introduction of Indexed Linked List data structure and the related algorithms.

7.1.1 Overview

The main objective of Stochastic Fair Drop policy is to minimize legitimate request drop rate and maximize malicious or misbehaving request drop rate. To achieve this goal, it considers all requests in the server's queue as a candidate to be removed from the queue whenever it is saturated. Assuming Q is the queue length, i.e., the number of requests in the queue, a **request** is an item in the queue whether it be a packet or an application request, then each request r_i in the queue has the following probability of being dropped for each dropping decision made under Stochastic Fair Drop policy:

$$P_d(r_i) = \frac{1}{Q}, \quad \forall\, 0 \leq i \leq Q. \tag{7.1}$$

Assuming there are $N_m(t)$ malicious requests and $N_l(t)$ legitimate requests in the queue at time t, the probability P_m of a malicious request being chosen for the drop at time t is given by

$$P_m(N_m, t) = \frac{N_m(t)}{N_m(t) + N_l(t)} = \frac{N_m(t)}{Q(t)}, \tag{7.2}$$

where $Q(t)$ is the queue length at time t. The probability P_l of a legitimate request being chosen for the drop is given by

$$P_l(N_l, t) = \frac{N_l(t)}{Q(t)}. \tag{7.3}$$

Stochastic Fair Drop policy can be enforced during the `ENQUEUE` operation of the queue, as shown in Algorithm 7.1.

The `QUEUE.DELETE` operation given in line 5 in Algorithm 7.1 requires a queue data structure that can provide efficient delete operation at any randomly chosen index, while maintaining the efficiency of existing enqueue and dequeue operations. Next, we propose a novel data structure that meets this requirement.

Algorithm 7.1: SFDQ.ENQUEUE(*request*)

```
1  QUEUE.ENQUEUE(request)
2  increment queue.length by 1
3  if queue.length == queue.capacity then
4      index = GETRANDOM(0, queue.length)
5      QUEUE.DELETE(index)
```

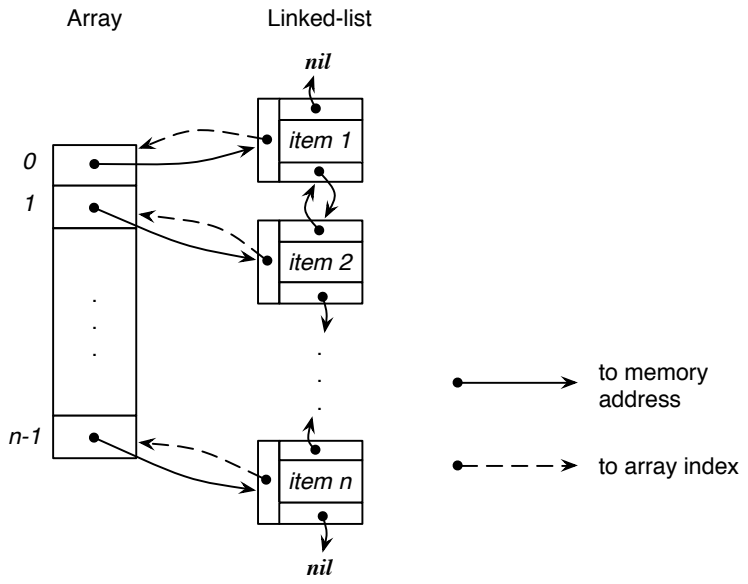


Figure 7.1: The Indexed Linked List data structure

7.1.2 Indexed Linked List

To efficiently implement Stochastic Fair Drop policy, we must implement constant time deletion of any randomly chosen element as well as implement constant time insertion and deletion at the two ends of the queue. A linked list implementation provides constant time deletion of an element if we're given a pointer to that element. However, we cannot get a pointer to a randomly chosen index in constant time. An array based implementation provides constant time access to a random element, but removing a random index takes $\Theta(\frac{n}{2})$ time on average, where n is the number of elements in the array.

To this end, we propose a novel data structure, called *Indexed Linked List*, to implement

the insertion and deletion operations in constant time. This hybrid data structure combines a linked list with an array of pointers to elements in the linked list. The actual data items are stored in the linked list, and each linked list node is pointed by an item in the array. Each entry in the array contains the address of the corresponding linked list node, and each linked list node contains the integer index of the corresponding array item. This bi-directional mapping relation between linked list nodes and array entries is illustrated in Figure 7.1. We can think of the array as an index over the linked list nodes.

The Indexed Linked List data structure implements three constant time operations: (1) ENQUEUE, (2) DEQUEUE, (3) DELETE; ENQUEUE adds the given item to the end of list; DEQUEUE extracts the item at the front of the list; and DELETE removes the item at a randomly given index, provided that the index is valid.

Algorithm 7.2: INDEXEDLINKEDLIST.ENQUEUE(*item*)

```
   Input: item - the new item to be enqueued
   Data: numItems - the number of items in the Indexed Linked List;
         capacity - the maximum capacity of the Indexed Linked List;
         linkedlist - the underlying linked list used by the Indexed Linked List;
         array - the underlying array used by the Indexed Linked List
1  if numItems ≥ capacity then
2     return false
   /* LINKEDLIST.PUSHBACK returns pointer to the linked-list node that stores item   */
3  node = LINKEDLIST.PUSHBACK(item)
4  array [numItems ] = node              /* saves the address of the new node in array */
5  node.index = numItems                 /* saves the matching array index in the node */
6  increment numItems by 1
7  return true
```

The ENQUEUE operation of Indexed Linked List is given in Algorithm 7.2. If the Indexed Linked List has not reached its maximum capacity, then the item is pushed back to the underlying linked list data structure. Next, the index or the array is updated to store the address of the linked list node where the new item is stored. As the pseudocode on line 4 of Algorithm 7.2 shows, address of the new node is always stored right after the current last element in the array. Meanwhile, the linked list node that stores the new item is updated to keep the index of the array item that points to it. At this point, a *bi-directional reference* is established. Using this bi-directional reference, we can get to the linked list node from the array index, and vice versa, in constant time. The bi-directional reference is crucial

to the Indexed Linked List's ability to randomly access and delete an item at any given index in constant time. When Stochastic Fair Drop policy uses Indexed Linked List as the underlying queue, the `QUEUE.ENQUEUE` operation in line 1 of Algorithm 7.1 simply becomes `INDEXEDLINKEDLIST.ENQUEUE`.

Algorithm 7.3: `INDEXEDLINKEDLIST.DEQUEUE`(*outItem*)

```
   Output: outItem - the dequeued item
1  if numItems == 0 then
2     return false
3  outItem = LINKEDLIST.FRONT()
   /* LINKEDLIST.POPFRONT returns the array index of stored in the popped node      */
4  index = LINKEDLIST.POPFRONT()
   /* SWAPLASTINDEXWITH swaps the last index with the given 'index'                 */
5  INDEXEDLINKEDLIST.SWAPLASTINDEXWITH(index)
6  decrement numItems by 1
7  return true
```

The `DEQUEUE` operation of Indexed Linked List is given in Algorithm 7.3. It first pops the first item in the linked list and saves it in the memory location passed to it. The linked list used by the Indexed Linked List also returns the index stored in popped node; note that this index is the reference from the linked list node to the array item in the bi-directional referenced that we described earlier. Removing the front node of the linked list leaves the array item pointing to that node to become an invalid reference, thus needs to be handled properly. The function call `SWAPLASTINDEXWITH` in line 6 of Algorithm 7.3 swaps this array item with the last array item and updates the linked list node that is pointing to the last array item to point to this array item, thereby getting rid of the invalid reference. Algorithm 7.4 describes the function `SWAPLASTINDEXWITH` in more detail.

Algorithm 7.4: `INDEXEDLINKEDLIST.SWAPLASTINDEXWITH`(*index*)

```
   /* This function swaps the last array item with the array item at the given index,
      and updates linked list node pointed by the last array item to point to the
      given index.                                                                  */
   Input: index - index of the array item to be swapped with the last array item
1  if numItems == 0 or index > numItems then
2     return
3  node = array [numItems- 1]  /* gets linked list node pointed by the last array item */
4  node.index = index             /* updates the node to point to the given 'index' */
5  array [index] = node          /* updates array item at 'index' to point to 'node' */
```

As shown in Algorithm 7.4, `SWAPLASTINDEXWITH` updates array item given by the in-

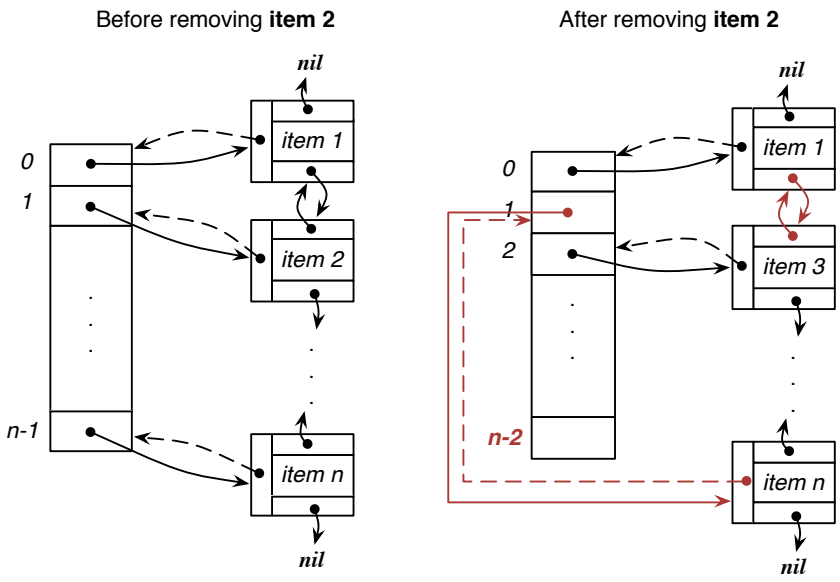


Figure 7.2: Removal of an item from the Indexed Linked List

put argument *index* to point to the linked list node pointed by the last array item, while also updating the that linked list node to point to the given *index*. In other words, `swapLastIndexWith` steals the linked list node of last array item and gives it to the array item at *index*. `swapLastIndexWith` is called only when an item is removed from the indexed linked list, that is why the last array item does not need to be updated as it is essentially being removed from the array via decreasing the array size by 1.

Algorithm 7.5: `IndexedLinkedList.delete`(*index*)

```
   Input: index - index of the item to be deleted
1  if numItems == 0 or index ≥ numItems then
2      return false
3  node = array [index]
4  LinkedList.remove(node)
5  IndexedLinkedList.swapLastIndexWith(index)
6  numItems −−
7  return true
```

The `delete` operation is given in Algorithm 7.5. It first removes the linked list node pointed by the array item at the given *index*, then swaps the last array item with the array item at the removed *index*. Figure 7.2 illustrates the `delete` operation by showing the

structure of a Indexed Linked List before and after an item is removed. When Stochastic Fair Drop policy uses Indexed Linked List as the underlying queue, the `QUEUE.DELETE` operation in Algorithm 7.1 becomes `INDEXEDLINKEDLIST.DELETE`.

Note that Indexed Linked List *preserves the insertion order* of items it contains, meaning if item A is enqueued before item B, then item A will always be dequeued before item B, unless it is being dropped before being dequeued. In this respect, Indexed Linked List resembles a First-In, First-Out (FIFO) queue.

7.2 DROP-BASED MISBEHAVIOR DETECTION & BLACKLISTING

Although Stochastic Fair Drop policy preferrentially drops requests from misbehaving users that are sending requests at rates far exceeding their fair share, there is still non-negligible chance of a legitimate request being dropped. Recall from Equation (7.3) that the probability of a legitimate request being dropped $P_l(N_l, t) = \frac{N_l(t)}{Q(t)}$. Let X be the random variable representing the number of drops until a legitimate request is being dropped, then the expected value of X is given by

$$\begin{aligned} E[X] &= \sum_{n=1}^{\infty} nPr[X = n] \\ &= \sum_{n=1}^{\infty} n(1 - P_l)^{n-1} P_l \\ &= \frac{1}{P_l}. \end{aligned} \tag{7.4}$$

For example, when $P_l(N_l, t) = 0.1$, we expect to drop a legitimate request for each 10 request being dropped. As long as the malicious requests keep arriving and filling up the queue, request dropping will continue to happen, and the legitimate requests will continously be at the risk of being dropped.

To further minimize the probability of legitimate requests being dropped, we adopt a simple but robust mechanism, called *Drop-based Misbehavior Detection & Blacklisting* (DMDB), to further minimize malicious requests from occupying the service queue. DMDB mechanism involves two steps:

1. The sender of a dropped request is labeled as a *suspect*, and be placed into a *suspect list* (SL) for the duration of t_S seconds; we call t_S the *suspicion interval*;
2. A suspect that has more than d_{BT} number of requests being dropped during the suspicion interval is regarded as malicious or misbehaving, and placed into a *blacklist* (BL) for t_B seconds; we call d_{BT} the *blacklist threshold*, and t_B the *blacklist interval*.

The rationale behind this two step approach is simple: misbehaving users have more requests in the queue and their requests are more likely to be dropped, therefore the owner of a dropped request can be considered *suspicious*; meanwhile, if a suspected user has multiple requests being dropped within a short period of time, that is a clear indication that it is sending requests at a rate exceeding its fair share, and should be penalized. The probability of a well-behaving user being wrongfully blacklisted under the DMDB mechanism can be minimized to a negligible level by setting the suspicion interval length and the blacklist threshold based on to the expected request rate of a well-behaving user.

Algorithm 7.6 describes the DMDB mechanism as part of the Stochastic Fair Drop Queue (SFDQ). Recall that SFDQ is implemented on top of the Indexed Linked List data structure that we introduced in previous section. DMDB uses the Auto Expire Cache data structure that we introduced in Section 4.2.2 of Chapter 4 for storing both the suspect list and the blacklist.

The modified `SFDQ.enqueue` operation first checks if the sender of a request is blacklisted before accepting the request into the service queue. It calls `DMDB.isBlackListed` operation to check for the blacklisting. `DMDB.isBlackListed` operation is fairly simple, it checks the blacklist, which is an Auto Expire Cache, to see if the given client ID (or sender ID) is in the list. Note that a current time value is passed to the `contains` method of the Auto Expire Cache, which uses it to automatically expire the old entries as we discussed in Section 4.2.2.

Another change to the `SFDQ.enqueue` operation is that it calls the `DMDB.handleSuspect` operation to notify a request drop if a drop does happen. `DMDB.handleSuspect` operation first checks if the given client ID is in the suspect list. If it is in the suspect list, it increases the drop count for that client ID; if it's not, it add the client ID to the suspect list. Next, the client ID is added to the blacklist if the updated drop count for it exceeds the blacklist threshold d_{BT}.

Algorithm 7.6: Stochastic Fair Drop Queue (SFDQ)

Data: indexedLinkedList - the underlying Indexed Linked List;
suspectList - Auto Expire Cache used as the *suspect list*;
blacklist - Another Auto Expire Cache used as the *blacklist*;
t_S - the *suspicion interval*;
t_B - the *blacklist interval*;
d_{BT} - the *blacklist threshold* (unit: number of request drops);
current_time - the timestamp of current time.

```
1  function Initialize()
2      suspectList.setExpirationInterval(t_S)
3      blacklist.setExpirationInterval(t_B)
4  function SFDQ.enqueue(request)
5      if DMDB.isBlackListed(request.clientId) then
6          return
7      indexedLinkedList.enqueue(request)
8      if indexedLinkedList.isFull() then
9          index = GetRandom(0, indexedLinkedList.numItems)
10         indexedLinkedList.delete(index)
11         DMDB.handleSuspect(request.clientId)
12 function DMDB.isBlackListed(clientId)
13     return blacklist.contains(clientId, current_time)
14 function DMDB.handleSuspect(clientId)
15     if suspectList.contains(clientId, current_time) then
           /* increase the drop count for clientId by 1                    */
16         suspectList.increaseCount(clientId, 1, current_time)
17     else
           /* add clientId to suspectList, and set its drop count to 1     */
18         suspectList.addItem(clientId, current_time)
19     if suspectList.getCount(clientId, current_time) ≥ d_BT then
20         blacklist.addItem(clientId, current_time)
```

All operations in SFDQ algorithm have constant $O(1)$ time complexities; and all operations in DMDB algorithm have amortized constant time $O(1)$ complexity due to the underlying Auto Expire Cache data structure's amortized $O(1)$ complexity.

7.3 EVALUATION AT THE APPLICATION LAYER

In this section, we evaluate the DDoS defense effectiveness of Stochastic Fair Drop Queue at the application layer.

7.3.1 Setup of Experimentation Environment

Overall setup of the experimentation environment is the same as the evaluation framework described in Section 3.3 of Chapter 3, with the additional configurations as follows.

- We only consider the flooding attacks where a malicious client sends requests at a rate that is m times the rate of a legitimate client; we call m the *attack multiplier*. We vary the attack multiplier m from 2 to 20 with an increment of 2. The percentage of attackers, also known as *attack intensity*, is varied from 10% to 90% with and increment of 10%.
- SFDQ is implemented in Network Simulator 2 (NS-2) by extending the generic Queue object and adding the stochastic fair drop and blacklisting behavior that are described in previous sections.
- Although SFDQ can be used under attacks that spoof unique identities, our experiments are limited to studying the DDoS attacks without spoofing. This is to better compare the effectiveness of SFDQ to other queuing mechanisms under equal terms. The DDoS defense efficacy of SFDQ with or without blacklisting is compared to that of the Deficit Round Robin (DRR) queue [66].
- The server's CPU rate f_{server} is configured using the following simple equation:

$$f_{server} = N \times \bar{\lambda}_{legit} \times \bar{C}_{req}, \tag{7.5}$$

where, N is the total number of clients, $\bar{\lambda}_{legit}$ is the average request rate of per legitimate client (unit: requests per second), and $\bar{C}_{req}$ is the average number of instructions to process one request. The reason for using such an equation is to derive a server CPU capacity that can be approximately saturated by the load generated by all clients combined in no attack scenario. Using this equation, the average server utilization ρ is measured around 61% when all clients are legitimate.

- The size of the service queue is configured using a simple equation as follows.

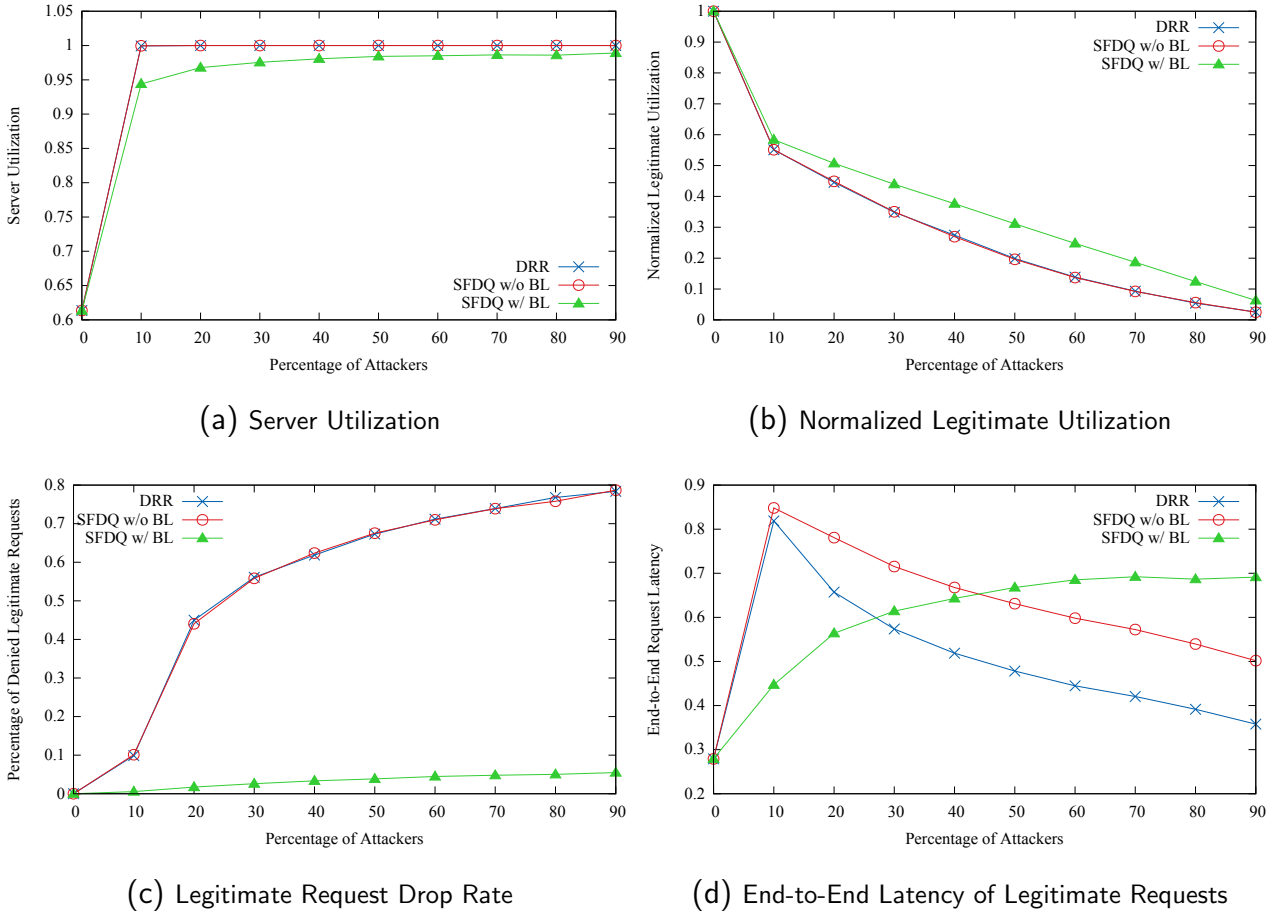
$$Q_{server} = \frac{N \times \bar{\lambda}_{legit}}{2} \tag{7.6}$$

  When studying the effect of the service queue size on the SFDQ blacklisting false positive rate, we manually configured it to take various values.
- The suspicion interval t_S is set to 1 second; the sizes of both the suspect list and the blacklist are set to 100; the blacklist threshold d_{BT} is set to 2; and lastly, the blacklist interval t_B is set to 120 seconds.
- In addition to measuring the utilization, request drop rate and latency, we also measure the false positive rate of the drop-based misbehavior detection mechanism of SFDQ.

7.3.2 Results

Three sets of results are obtained as following: (1) vary the percentage of malicious clients, or attack intensity, while fixing the total number of clients; (2) vary the attacker multiplier while fixing the attack intensity to 50%; (3) vary the service queue size, while fixing other parameters. For the first and second set of experiments, the queue size is determined using Equation (7.6).

The first set of results are shown in Figure 7.3. The server utilization ρ is always close to 100% under all different attack intensity regardless of the queue being used, as plotted by Figure 7.3a. When SFDQ is used with its blacklisting mechanism turned on, it can keep the utilization rate slightly below the 100% utilization rate. This is because the blacklisting mechanism of SFDQ filters large portion of the request floods from the malicious clients. The normalized legitimate utilization $\tilde{\rho}_{legit}$ in Figure 7.3b shows that SFDQ actually performs



(a) Server Utilization

(b) Normalized Legitimate Utilization

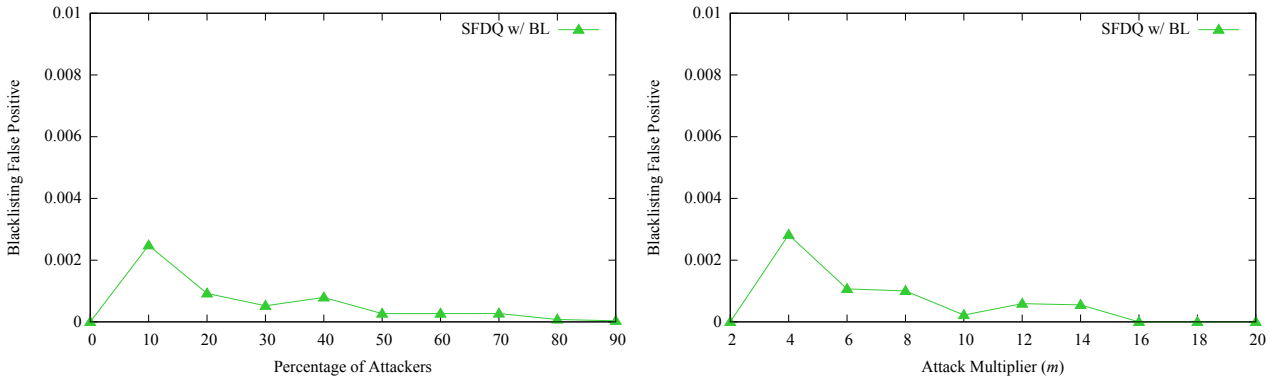
(c) Legitimate Request Drop Rate

(d) End-to-End Latency of Legitimate Requests

Figure 7.3: Performance of SFDQ under DDoS attack with varied attack intensity

better than DRR fair queuing in terms of protecting the fair share of legitimate clients. It may seem that the legitimate utilization is not proportional to the ratio of legitimate clients in system when SFDQ is used. This is because the server is only 61% utilized when all clients are malicious, and the malicious clients take up more of the server's spare capacity than do the legitimate clients.

The percentage of legitimate requests that are denied is plotted in Figure 7.3c. Using SFDQ without the blacklisting leads to significant subset of the legitimate requests being dropped as expected, while less than 5% of legitimate requests being dropped when SFDQ is used with blacklisting switched on. The legitimate request drop rate can be further



(a) Varied Attack Intensity

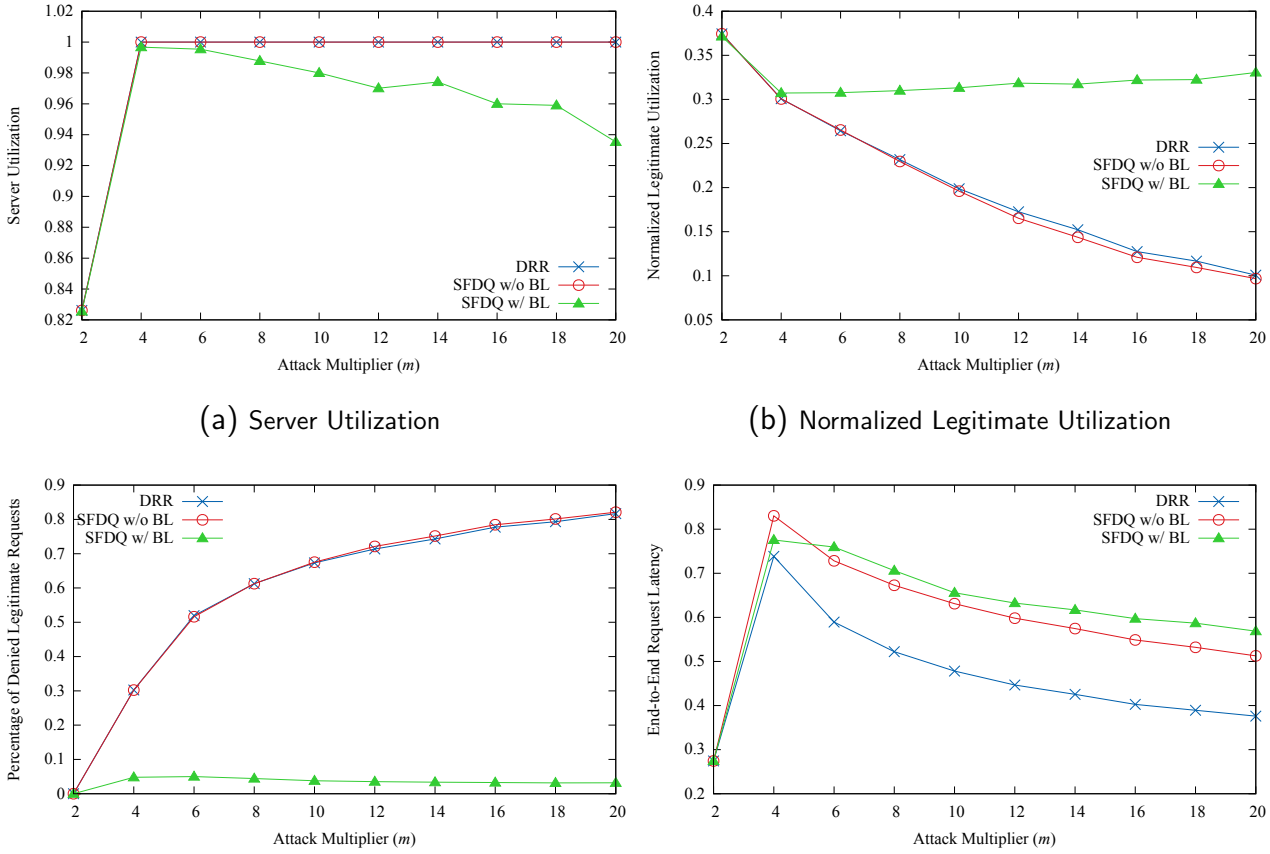
(b) Varied Attack Multiplier

Figure 7.4: SFDQ blacklisting false positive rate

minimized if the blacklisting duration is increased, which is safe to do as the blacklisting false positive rate of SFDQ is very low as shown in Figure 7.4.

SFDQ also keeps the average end-to-end latency of legitimate requests fairly low and stable across different attack intensity, as shown in Figure 7.3d. An interesting trend that can be observed in Figure 7.3d is that the average latency of legitimate requests decreases as the attack intensity increases when DRR and SFDQ without blacklisting is used. This is because of the decreased throughput of legitimate requests due to high drop rate.

Overall, SFDQ performs extremely well against DDoS attacks of varying attack intensity across all metrics we measure. The low false positive rate of the blacklisting plays a critical role in the overall effectiveness of SDFQ. The false positive rate r_{fp} is calculated as $r_{fp} = \frac{B_g}{B_{total}}$, where B_g is the number of times a legitimate client is blacklisted and B_{total} is the total number of blacklisting decisions made. The false positive rates are very low across all attack intensities, as shown in Figure 7.4a, and across all attack multipliers, as plotted in Figure 7.4b. As shown in Figures 7.4a and 7.4b, the false positive rate decreases as the attack intensity increases or the attack multiplier increases. This is expected, because the difference between the request drop rates of legitimate clients and malicious clients becomes larger as the malicious clients become more aggressive, thus enabling the drop-based misbehavior



(a) Server Utilization

(b) Normalized Legitimate Utilization

(c) Legitimate Request Drop Rate

(d) End-to-End Latency of Legitimate Requests

Figure 7.5: Performance of SFDQ under DDoS attack with varied attack multiplier

detection to more accurately identify misbehavior. As Figure 7.4 shows, SFDQ has a false positive rate that is below 0.003 even in the worst case.

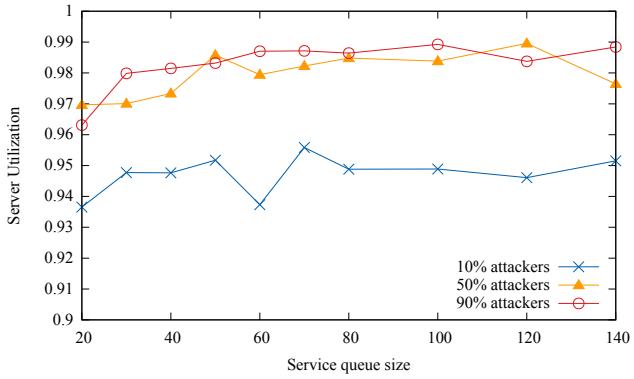
The second set of experiment results, where the attack multiplier m is varied, are shown in Figure 7.5. The attack intensity (or, the percentage of malicious clients) is fixed at 50% for these experiments. SFDQ still performs well across all attack multiplier settings for all metrics we measured, allowing only a negligible fraction of the legitimate requests to be affected by the attacks. The performance of DRR and SFDQ without blacklisting is as expected, and similar to the results in Figure 7.3. The results for SFDQ with blacklisting are different from the results in Figure 7.3 in several ways. First, the utilization of the server

ρ and the normalized legitimate utilization $\tilde{\rho}_{legit}$ both decrease after attack multiplier $m = 4$ when SFDQ with blacklisting is used. This can explained by the fact that the time it takes to detect a misbehavior shortens as the drop rate per second of malicious requests increases due to the increasingly higher request rate of malicious clients. SFDQ filters out a lot more of the malicious requests due to this faster detection speed, saving more capacity of the server from the malicious clients. This gain in the server capacity saved from malicious clients is then used by the legitimate clients, and leading to the slight increase in the normalized legitimate utilization as the attack multiplier increases. The normalized legitimate utilization of the server is plotted in Figure 7.5b.

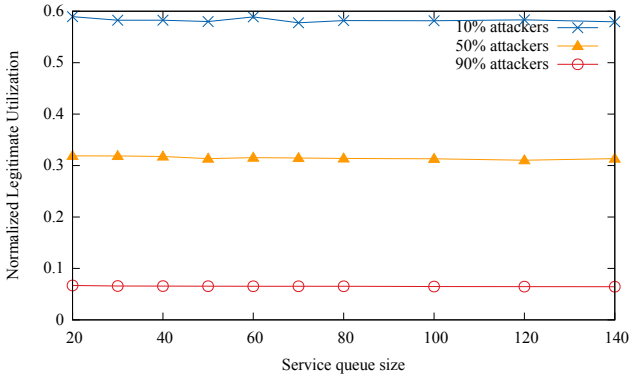
The percentage of denied legitimate requests and the average end-to-end latency of legitimate requests both decrease as the attack multiplier increases when SFDQ with blacklisting is used, as shown in Figure 7.5c and 7.5d respectively. This decrease can similarly be explained by the same observation that the SFDQ detects the misbehavior faster as the misbehavior gets more aggressive.

Lastly, we evaluate SFDQ with blacklisting under varied service queue size configurations to measure the effect of service queue size on SFDQ performance. As we expected, the queue size does not have a significant impact on the performance SFDQ, as shown in Figure 7.6. The share of the server's capacity used for legitimate requests does not change due to service queue size changes, as shown in Figure 7.6b. As the size of the service queue increases, the percentage of dropped legitimate requests decrease slightly, while the average end-to-end latency of legitimate requests increase significantly, shown in Figure 7.6c and 7.6d respectively. Increased latency and decreased request drop rate with respect to the increased queue size are the inherent properties of a queue, and thus are not a side-effect of decreased or increased effectiveness of SFDQ.

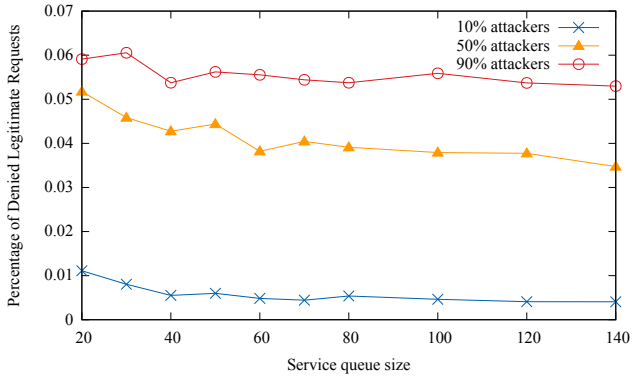
Figure 7.6e shows that the blacklisting false positive rate increases as the size of the service queue increases. However, the increase in the false positive is very limited — about less than 6%. The slight increase in false positive rate can be explained as follows. As the queue size increases, the probability of a request being dropped decreases due to the inherent property of a queue; as such the drop rate of malicious requests also slightly decreases, making it slightly harder for the SFDQ to detect the misbehavior. This explanation is supported by



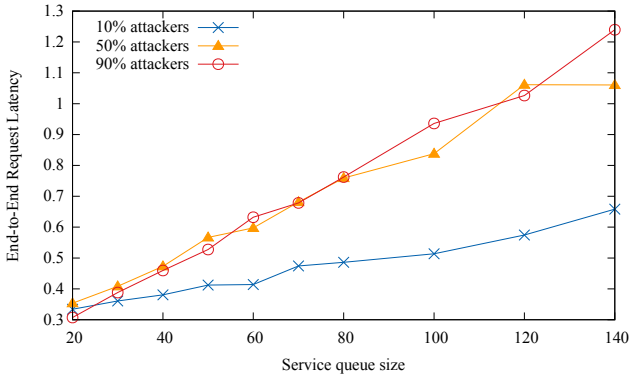
(a) Server Utilization



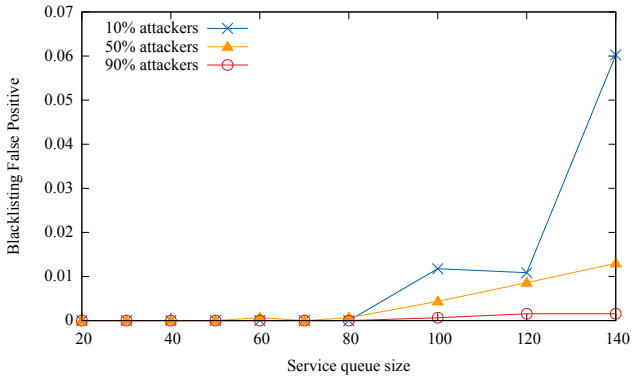
(b) Normalized Legitimate Utilization



(c) Legitimate Request Drop Rate



(d) End-to-End Latency of Legitimate Requests



(e) Blacklisting False Positive Rate

Figure 7.6: Effect of service queue size on SFDQ performance

the fact that the false positive rate under the 90% attack intensity is almost wasn't affected by the change in service queue size since the malicious request drop rate stays very high at that attack intensity without being significantly affected by the queue size change.

7.4 EVALUATION AT THE NETWORKING LAYER

Although SFDQ is originally motivated by defending against application layer attacks, it can be applied to defending against bandwidth flooding attacks at the network layer. In this section, we show that SFDQ works equally well against network layer flooding attacks and compare its effectiveness against that of fair queuing. Although we do not provide a detailed study of SFDQ against flooding attacks that utilize IP address spoofing, we believe it is significantly less affected than fair queuing approaches as SFDQ uses IP address only to filter traffic whereas fair queuing critically depends on the accountability of IP addresses to allocate resources fairly.

7.4.1 Setup of Experimentation Environment

Our simulation setup for evaluation of SFDQ against network layer flooding is significantly different from the evaluation framework described in Section 3.3. First, we use a hybrid network topology, visualized in Figure 7.7, that is different from the two previously used topologies in our experiments. Similar topologies are used in several fair queuing and DDoS attack studies, such as [45] and [77].

The topology consists of 4 core routers in the middle that connects $192+N_b$ edge routers, where N_b is the number of edge routers used by the attack traffic. The bottleneck link is between the 3rd and 4th core routers, as shown in Figure 7.7. The DDoS attack is directed at flooding the bandwidth of this bottleneck link. The good traffic that go through this bottleneck link is sent from 32 edge routers that are connected to 3rd core router to another set of 32 edge routers that are connected to the 4th core router. These edge routers are denoted by a square in Figure 7.7. The rest of the edge router are used for generating the

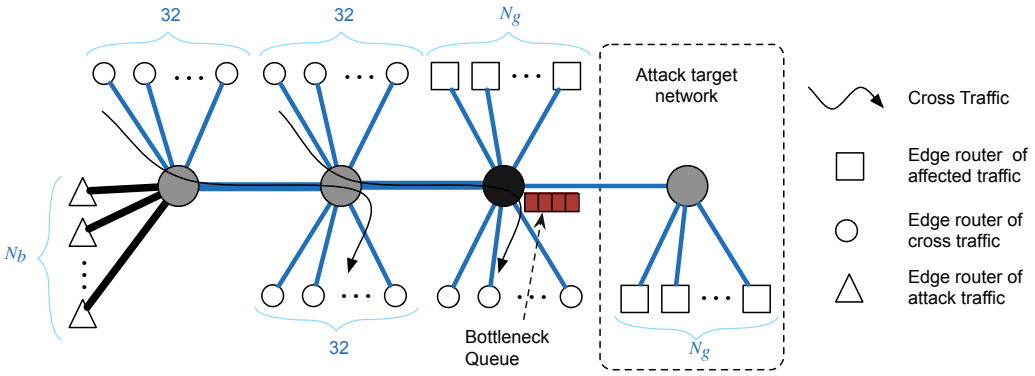


Figure 7.7: Topology used in network flooding attack experiments

background traffic that go through core routers 1 to 3, but not the bottleneck link.

The link bandwidth and delay of various routers are configured as follows. The links between core routers have a bandwidth of 1500 Mb/sec and a delay of 5 ms, except the bottleneck link, which has a bandwidth of 50 Mb/sec and a delay of 0.2 ms. All edge routers have a 50 Mb/sec bandwidth and a 10 ms delay, except the attacker's edge routers, which have 1500 Mb/sec bandwidth and a 5 ms link delay.

The packet queues of all core routers are set to use Droptail queues, except the bottleneck queue, which is configured to use SFDQ or DRR depending on which queuing mechanism we are experimenting with. The sizes of all queues, for edge and core routers alike, are set to 1,460,000 bytes, or 1000 packets since TCP [59] packet size is fixed to be 1460 bytes.

Long-lived TCP flows are attached to the edge routers, except for the edge routers of attack traffic. The maximum rate of each flow is configured to be 1.5 Mb/sec, whereas the attack traffic uses UDP [58] protocol, and sends at m times the rate of the benign TCP flows. TCP New Reno [33] version of the TCP protocol is used by all TCP flows.

SFDQ parameters are set as following: the suspicion interval t_S is set to 1 second; the sizes of both the suspect list and the blacklist are set to 100; the blacklist threshold d_{BT} is set to 10 when it is not varied; and the blacklist interval t_B is set to 30 seconds.

For all experiments, we measure the average throughput per legitimate TCP flows, the

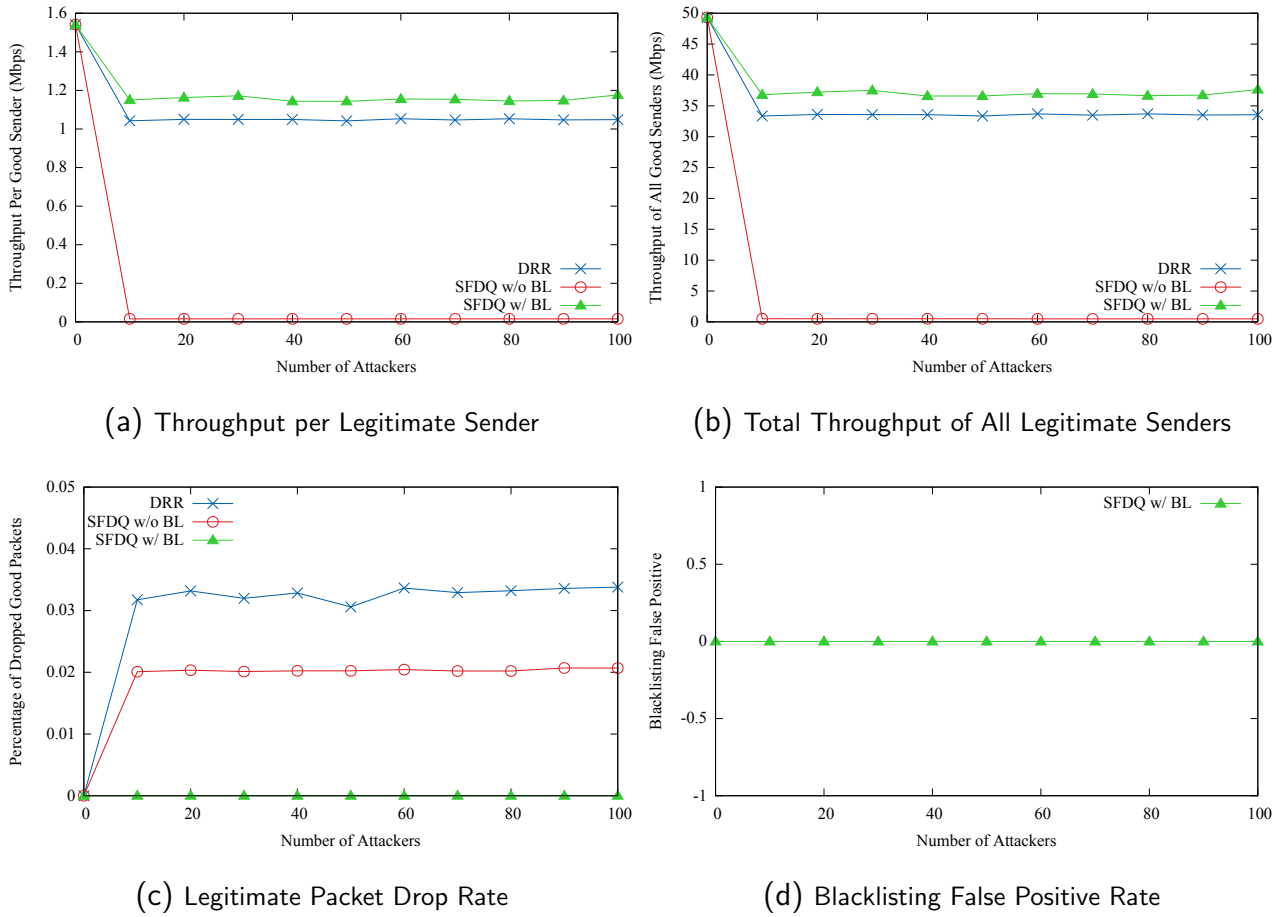


Figure 7.8: Effectiveness of SFDQ against network layer DDoS attacks with varied attack intensity

total throughput of all legitimate TCP flows, and the average packet drop rate of all legitimate TCP flows. For experiments that use SFDQ, we also measure the blacklisting false negative rate.

7.4.2 Results

The first set of results, shown in Figure 7.8, are of the experiments where the number of attack sources are varied from 10 to 100 with an increment of 10. The attack multiplier m is set to 10 in these experiments.

The average throughput per legitimate sender is plotted in Figure 7.8a. As shown, SFDQ

with blacklisting achieves a per legitimate sender throughput of around 1.2 Mb/sec, which is fairly close to the average throughput of 1.5 Mb/sec when there is no attack. As a consequence, the total throughput achieved by all legitimate senders when SFDQ is used is closer to the total throughput of 50 Mb/sec for the configuration with no attackers, as shown in Figure 7.8b.

The percentage of dropped packets for legitimate flows are fairly low across for both SFDQ and DRR, as shown in Figure 7.8c. This is mostly due to the congestion control mechanisms built into the TCP protocol. However, it is worth noting that SFDQ achieves almost zero packet drop rate for legitimate flows. The false positive rate of the SFDQ blacklisting mechanisms is equal to zero across all attack intensity, as shown in Figure 7.8d. This is expected as the misbehaving flows can be detected very precisely since they are highly likely to be dropped compared to the well behaved TCP flows due to their unrestrained sending rate.

The next set of experiments are conducted to measure the effectiveness of SFDQ with a setup where the attack flows are varied to take different rate, from well-restrained to highly unrestrained. The results are plotted in Figure 7.9.

The average per legitimate flow throughput and the total throughput of all legitimate flows are given in Figure 7.9a and 7.9b respectively. As we can see SFDQ with blacklisting guarantees fairly high throughput rates for legitimate flows. However, both the per legitimate flow throughput and total legitimate throughput decrease as the attack multiplier m increases. This is due to the fact that the amount of attack packets pass through the bottleneck router during the time between an attack source is not blacklisted to when it is blacklisted differs for different attack multiplier values: more attack packets pass through the bottleneck router during this time for a larger value of m than for a smaller value of m due to the higher packet rate for larger m. Thus, it increases the chance of legitimate flows being more affected due to higher congestion. There are two ways to address this issue. The first approach is to decrease the blacklist threshold d_{BT} so that SFDQ responds more quickly to misbehaving flows. The second approach is to increase the blacklist interval t_B so that the relative length of non-blacklisted periods for misbehaving flows are decreased significantly. We believe the second approach is more effective given the same false positive rate.

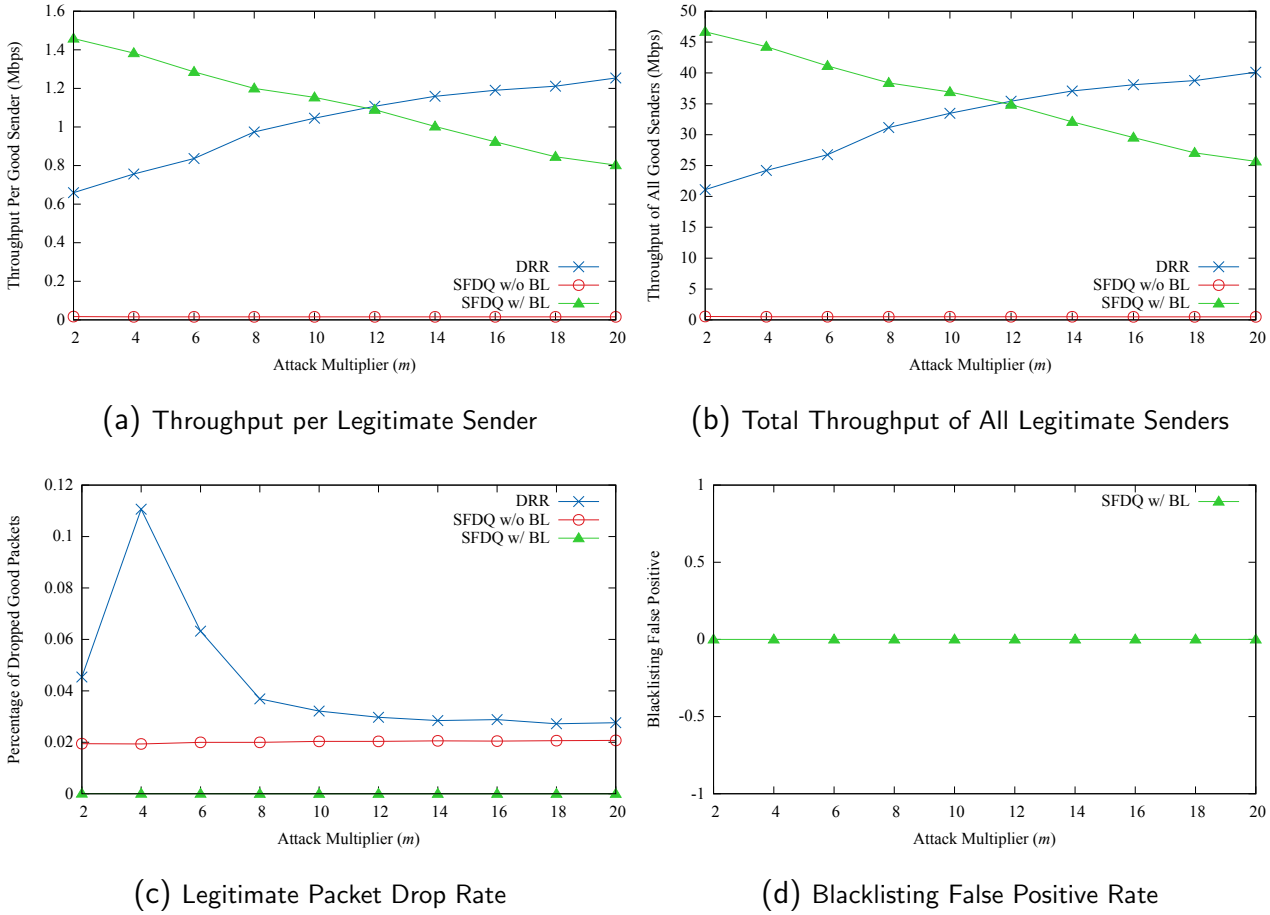
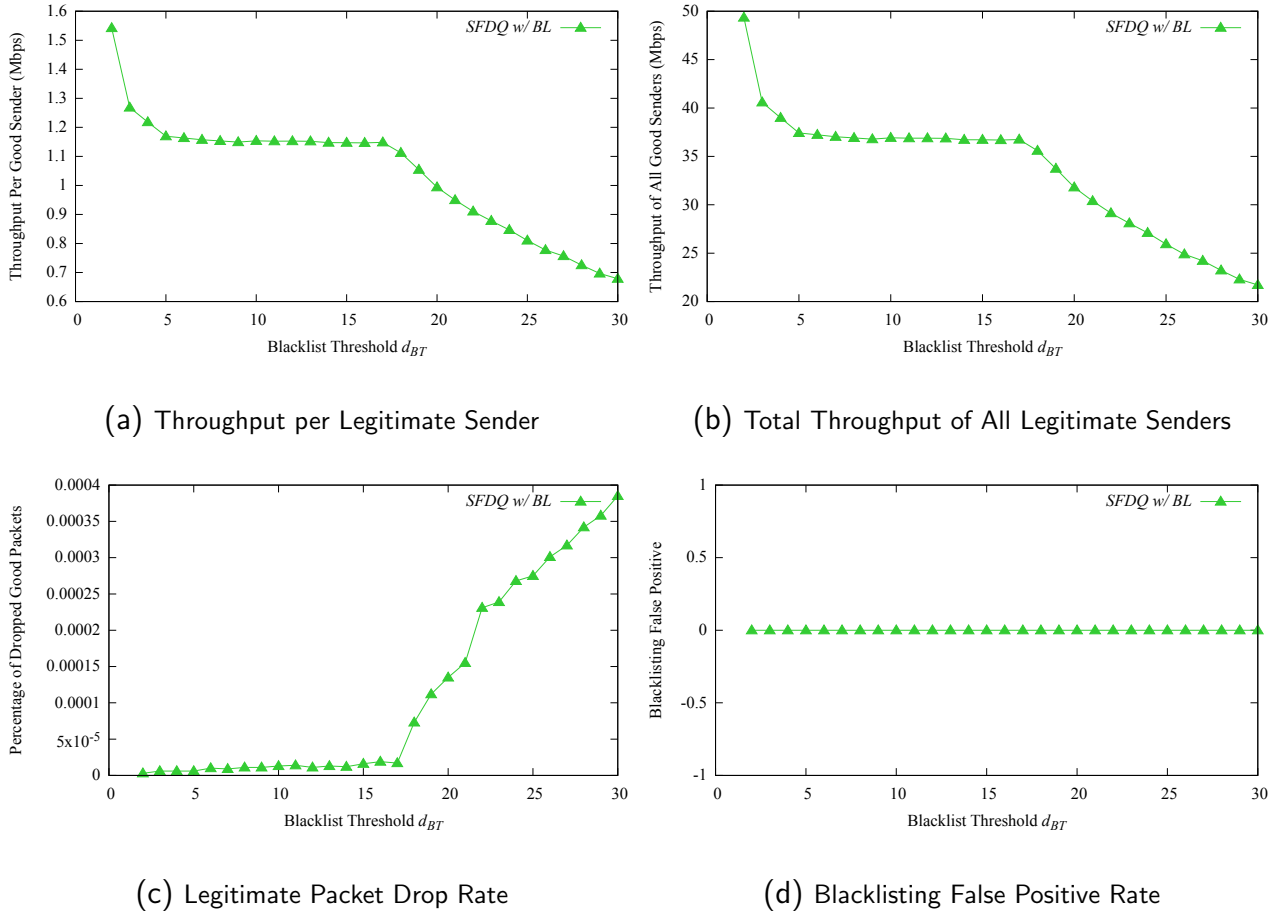


Figure 7.9: Effectiveness of SFDQ against network layer DDoS attacks with varied attack multiplier

The last set of experiments are conducted for varied values of the blacklisting threshold d_{BT} between 2 and 30, while fixing the number of attack sources N_b to 32 and the attack multiplier m to 10. The results are reported in figure 7.10. The immediate observation can be made is that the blacklisting false positive rate stays at zero, as plotted by Figure 7.10d, for all blacklisting threshold values chosen in these experiments. This shows that SFDQ's blacklisting mechanisms can be configured easily, and it is not very difficult to find a right value for the blacklisting threshold. This observation is further supported by the results in Figures 7.10a, 7.10b, and 7.10c. For the a fairly wide range of threshold values between 5 and 17, SFDQ provides about the same legitimate flow throughput and legitimate package



(a) Throughput per Legitimate Sender

(b) Total Throughput of All Legitimate Senders

(c) Legitimate Packet Drop Rate

(d) Blacklisting False Positive Rate

Figure 7.10: Effectiveness of SFDQ with varied blacklisting threshold against network layer DDoS attacks

drop rate.

7.5 CONCLUSION

In this chapter, we proposed a request dropping mechanism, called Stochastic Fair Drop, that probabilistically drops a request out of all the requests when the queue is full. This new queue management approach is motivated by the simple observation that there are significantly more malicious requests than the legitimate requests reside in a queue that

is experiencing DDoS attacks, thus dropping items in the queue at random leads to more malicious requests being dropped. Following this first observation, we made the second observation that misbehavior can be detected with high accuracy by monitoring only the senders of requests that are dropped under the Stochastic Fair Drop policy. We designed a simple yet robust detection and blacklisting mechanism, called Drop-based Misbehavior Detection & Blacklisting (DMDB), based on the second observation.

To efficiently implement the Stochastic Fair Drop policy, we introduced a novel data structure, called Indexed Linked List, that provides enqueue, dequeue, and remove operations with $O(1)$ time complexity. We provided the algorithms for a Stochastic Fair Drop Queue (SFDQ) that incorporates the fair drop policy as well as the DMDB mechanism. Through extensive simulation study of SFDQ against the DDoS attacks at both the application layer and the network layer, we showed the SFDQ provides strong defense against DDoS attacks while remaining simple and easy to configure.

8.0 THESIS SUMMARY & CONCLUSION

In this chapter, we provide a summary of our dissertation study as well as the contributions we made, followed by the conclusion of the entire thesis.

8.1 SUMMARY OF RESULTS AND CONTRIBUTIONS

DDoS attacks pose a great threat to the Internet and its public services. In particular, service level or application layer denial of service attacks are increasing in size, sophistication, and operational impact. However, traditionally most DDoS research emphasis is placed on mitigating flooding DDoS attacks that target the downlink bandwidth of the victim. The known application layer DDoS defense solutions often use detection then mitigation strategy. But due to the difficulty of distinguishing attack from legitimate traffic with high accuracy, detection mechanisms are limited in their effectiveness.

In this dissertation, we investigated the application layer DDoS attacks against public Internet services, and designed and evaluated four different defense frameworks that do not rely on traditional detection methods.

We started out by analyzing the strength and weaknesses of a promising DDoS resistance mechanisms — puzzle based DDoS defense. On the plus side, puzzle based defense is not susceptible to DDoS attacks that leverage unique identity spoofing techniques, and tips the balance between the request overhead of the client and the response overhead of the server in favor of the server. On the other hand, we showed that existing puzzle based DDoS defense solutions are limited in their effectiveness against DDoS attacks and are wasteful of the clients' computational resources. The limited effectiveness of puzzle based DDoS defense

solutions is due to the fact that they do not provide dynamic determination of per-request puzzle hardness, while also lacking mechanisms to prevent or deter replay attacks. To this end, we proposed *Puzzle+* framework.

The hardness model proposed in Puzzle+ determines the puzzle hardness on per-request basis, taking into account the relative load that the request creates on the server's overall load. The model also considers the current load of the server when computing the puzzle hardness, and provides ability to control the server load at a given target threshold. In addition, Puzzle+ implements an efficient replay attack prevention mechanism that uses a novel caching algorithm, called Auto Expire Cache. Through extensive evaluation of the Puzzle+ framework, we showed that it provides significantly better guarantees to legitimate users of the system than the existing fixed puzzle hardness based schemes in terms of three key metrics — the legitimate utilization of the server, the percentage of denied legitimate requests, and the average end-to-end latency of legitimate requests.

The existing evaluations of puzzle based DDoS defense effectiveness assumed that client machines have approximately the same computational power. That raised concern among the researchers about the effectiveness of puzzle schemes when there is large disparity in the computational resources of client machines. To address such concerns, we studied Puzzle+ under situations where there is a large disparity in the client machine CPU speed. We found that Puzzle+ maintains the same level of effectiveness against DDoS attacks when the client CPU frequencies are randomly distributed following a normal distribution or a uniform distribution and the ratio of the fastest client CPU speed to the slowest varies between 10 and 100. On the other hand, we also found that Puzzle+ and other computation based puzzle schemes become significantly less effective when the disparity in CPU speed is not randomly distributed and all malicious client machines 10 to 100 times more powerful than any legitimate client machine.

Next, we argued that existing computational puzzle schemes are wasteful of client's computational resources, in that the puzzle computations do not contribute to solving any meaningful problem. To eliminate or alleviate the wastefulness, we proposed two novel puzzle schemes *Productive Puzzles* and *Guided Tour Puzzles*.

Productive puzzles transform the wasteful computations required by computational puz-

zles into computation of meaningful tasks that provide utility. We introduced *Known-Unknown tests* for the productive puzzles to detect cheating clients with a very high probability, and proved an upper-bound on the probability of successful cheating. To further minimize the chances of incorrect or bogus solutions being accepted, we incorporated the majority voting based redundancy mechanism with the Known-Unknown Tests. We showed that a very low error rate can be achieved using fairly small redundancy when majority voting is combined with the Known-Unknown Tests. Through extensive simulation study, we showed that the cost of computing the known tasks in productive puzzles is justifiable by the gain we get through the completion of unknown tasks, as the gains are significantly larger. Our experiment results supported the per-task error bounds that we derived mathematically. While minimizing the wasteful work, productive puzzles maintained the same level of effectiveness that provided by Puzzle+ against DDoS attacks.

To eliminate the wasteful computations imposed by traditional puzzle schemes entirely, we explored the novel idea of network delay based puzzles. We introduced Guided Tour Puzzles, a novel delay based puzzle scheme, and showed that it not only achieves the previously defined requirements of an effective puzzle scheme, but also it is not affected by the disparity in the client computational powers. Furthermore, we showed that Guided Tour Puzzles eliminate all wasteful computations at the client and do not burden the clients with additional computational or bandwidth requirements. By measuring the tour delays on an actual network test bed environment, we showed that the variation in the tour delays are smaller in comparison with the disparity in the computational powers. More importantly, we showed that tour delay variation cannot be effectively manipulated by malicious clients to achieve unfair advantage over legitimate clients. Meanwhile, using extensive simulation studies we showed that guided tour puzzle is indeed effective in mitigating distributed denial of service attacks, and that it is a practical solution to be adopted.

Last, but not least, we investigated novel DDoS defense mechanisms that can provide resilient defense at different layers of the Internet protocol stack. We made the observation that a common characteristic of the system under a flooding based DDoS attack is that the queue for storing the incoming requests contains significantly more malicious requests than legitimate requests. Based on this observation, we proposed a request drop policy called

Stochastic Fair Drop that randomly chooses the dropped request out of all requests in the queue. Following the first observation, we made the second observation that misbehavior can be detected with high accuracy by monitoring only the senders of requests that are dropped under the Stochastic Fair Drop policy. We designed a simple yet robust detection and blacklisting mechanism, called Drop-based Misbehavior Detection & Blacklisting (DMDB), based on the second observation.

To efficiently implement the Stochastic Fair Drop policy, we introduced a novel data structure, called Indexed Linked List, that provides enqueue, dequeue, and remove operations with $O(1)$ time complexity. We provided the algorithms for a Stochastic Fair Drop Queue (SFDQ) that incorporates the fair drop policy as well as the DMDB mechanism. Through extensive simulation study of SFDQ against the DDoS attacks at both the application layer and the network layer, we showed the SFDQ provides strong defense against flooding based DDoS attacks while remaining simple and easy to configure.

8.2 CONCLUSION

The puzzle based DDoS defense is a promising approach to mitigating DDoS attacks in the Internet. In order to provide effective defense against DDoS attacks, however, they must satisfy various efficiency and security requirements. Among others, an accurate and dynamic determination of puzzle hardness on a per-request basis as well as the ability to prevent puzzle solution replay attacks are critical for the success of any puzzle based DDoS defense framework. Puzzle+ — the improved computational puzzle framework that we proposed — satisfies both of these key requirements.

On the other hand, computation based puzzle mechanisms are wasteful of the client computational resources. Productive Puzzles alleviates the wastefulness, and puts the puzzle computations to good use by utilizing them towards solving meaningful computational problems. Not only do Guided Tour Puzzles eliminate the wasteful computations required by the computational puzzles, but also they are significantly less affected by the disparity in the computational resources of the client machines that perform the puzzle computations.

Both of these novel puzzle frameworks achieve effective mitigation of DDoS attacks while satisfying the no wasteful computation requirement.

DDoS attacks must be defended at all layers of the protocol stack that are vulnerable. Stochastic Fair Drop Queue is a novel queuing mechanism that is simple, efficient, and easy to use. It is built on the observation about the key property of a queue that is experiencing DDoS attacks. It provides a strong defense at the application layer, and a strong resilience against DDoS attacks at the network layer.

BIBLIOGRAPHY

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. In *10th Network and Distributed System Security Symposium*, pages 25–39, 2003.

[2] Mehmud Abliz. Internet Denial of Service Attacks and Defense Mechanisms. Technical Report TR-11-178, University of Pittsburgh Department of Computer Science, March 2011. Available online: http://www.cs.pitt.edu/~mehmud/docs/abliz11-TR-11-178.pdf.

[3] Mehmud Abliz and Taieb Znati. New Approach to Mitigating Distributed Service Flooding Attacks. In *the 7th International Conference on Systems (ICONS '12)*, Reunion Island, 2012.

[4] T. Aura, P. Nikander, and J. Leiwo. DoS-resistant authentication with client puzzles. In *8th International Workshop on Security Protocols*, volume 2133, pages 170–181, 2000.

[5] Adam Back. Hashcash - A Denial of Service Counter-Measure, 2002.

[6] Hitesh Ballani and Paul Francis. Mitigating dns dos attacks. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 189–198, 2008.

[7] Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. *SIGMETRICS Perform. Eval. Rev.*, pages 151–160, 1998.

[8] Jon C. R. Bennett and Hui Zhang. Hierarchical Packet Fair Queueing Algorithms. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, pages 143–156, New York, NY, USA, 1996. ACM.

[9] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), January 2005.

[10] Matt Bishop. *Computer Security: Art and Science*, chapter 1, pages 3–6. Addison Wesley, 2002.

[11] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[12] Nikita Borisov. Computational puzzles as sybil defenses. In *P2P '06: Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing*, pages 171–176, Washington, DC, USA, 2006.

[13] Ronald H. Brown and Arati Prabhakar. Secure hash standard, 1995.

[14] CERT/CC. Tcp syn flooding and ip spoofing attacks. CERT Advisory CA-1996-21, September 1996.

[15] CERT/CC. Denial of service attacks, October 1997.

[16] CERT/CC. Cert statistics (historical), February 2009.

[17] David D. Clark. The design philosophy of the DARPA internet protocols. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 106–114, New York, NY, USA, 1988.

[18] M. J. Coster, A. Joux, B. A. Lamacchia, A. M. Odlyzko, C.P. Schnorr, and J. Stern. Improved low-density subset sum algorithms. *Computational Complexity*, 2(2), 1992.

[19] D. Dean and A. Stubblefield. Using client puzzles to protect TLS. In *10th USENIX Security Symposium*, pages 1–8, 2001.

[20] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '89, pages 1–12, New York, NY, USA, 1989. ACM.

[21] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[22] Roland Dobbins and Carlos Morales. Worldwide Infrastructure Security Report. Arbor Networks Annual Survey, December 2010.

[23] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In *CRYPTO '03*, 2003.

[24] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO '92*, pages 139–147, 1992.

[25] W. Feng. The case for TCP/IP puzzles. In *ACM SIGCOMM Future Directions in Network Architecture*, 2003.

[26] W. Feng, E. Kaiser, and A. Luu. The design and implementation of network puzzles. In *IEEE INFOCOM '05*, volume 4, pages 2372–2382, Miami, FL, 2005.

[27] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin Backbone Protocol: Analysis and Applications. Technical report, International Association for Cryptologic Research, 2014.

[28] Lawrence A. Gordon, Martin P. Loeb, , William Lucyshyn, and Robert Richardson. CSI/FBI computer crime and security survey. Annual Report, 2005.

[29] B. Groza and D. Petrica. On chained cryptographic puzzles. In *3rd Romanian-Hungarian Joint Symposium on Applied Computational Intelligence (SACI '06)*, Timisoara, Romania, 2006.

[30] Bogdan Groza and Bogdan Warinschi. Cryptographic Puzzles and DoS Resilience, Revisited. *Designs, Codes and Cryptography*, 73(1):177–207, 2014.

[31] M. Handley, E. Rescorla, and IAB. Internet Denial-of-Service Considerations. RFC 4732 (Informational), December 2006.

[32] J. Heinanen and R. Guerin. A Single Rate Three Color Marker. RFC 2697 (Informational), September 1999.

[33] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582 (Proposed Standard), April 2012.

[34] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *the IFIP TC6/TC11 Joint Working Conference on Secure Information Networks*, pages 258–272, 1999.

[35] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *NDSS '99*, pages 151–165, San Diego, CA, 1999.

[36] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997.

[37] Planet Lab. About planet lab. http://www.planet-lab.org/about.

[38] Craig Labovitz. Botnets, DDoS and Ground-Truth – A Look at 5,000 Confirmed Attacks. NANOG Presentation, October 2010.

[39] Craig Labovitz. Round 2: DDoS Versus Wikileaks. Web post, November 2010. http://asert.arbornetworks.com/2010/11/round2-ddos-versus-wikileaks/.

[40] Karthik Lakshminarayanan, Daniel Adkins, Adrian Perrig, and Ion Stoica. Taming IP Packet Flooding Attacks. *ACM SIGCOMM Computer Communication Review*, 34(1):45–50, 2004.

[41] Ben Laurie and Richard Clayton. "Proof-of-Work" Proves Not to Work. In *in WEAS 04*, 2004.

[42] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking (TON)*, 2(1):1–15, 1994.

[43] Robert Lemos. Web worm targets white house. CNET News, July 2001. http://news.cnet.com/2100-1001-270272.html.

[44] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[45] Jörg Liebeherr and Nicolas Christin. JoBS: Joint Buffer Management and Scheduling for Differentiated Services. In *Proceedings of the 9th International Workshop on Quality of Service*, IWQoS '01, pages 404–418, London, UK, UK, 2001. Springer-Verlag.

[46] Howard F. Lipson. Tracking and tracing cyber-attacks: Technical challenges and global policy issues. Special report CMU/SEI-2002-SR-009, Cert Coordination Center, November 2002.

[47] M. Ma. Mitigating denial of service attacks with password puzzles. In *International Conference on Information Technology: Coding and Computing*, volume 2, pages 621–626, Las Vegas, 2005.

[48] Steven McCanne, Sally Floyd, and Kevin Fall. NS-2 (network simulator 2).

[49] Andy McCue. 'Revenge' hack downed US port systems. ZDNet News, October 2003. http://www.zdnet.co.uk/news/security-management/2003/10/07/revenge-hack-downed-us-port-systems-39116978/.

[50] P.E. McKenney. Stochastic Fairness Queueing. In *INFOCOM '90, Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. The Multiple Facets of Integration. Proceedings, IEEE*, volume 2, pages 733–740, Jun 1990.

[51] Ron Meyran. WikiLeaks Attack Anatomy - Operation Payback. Technical Report, December 2010.

[52] Jelena Mirkovic, Sven Dietrich, David Dittrich, and Peter Reiher. *Internet Denial of Service: Attack and Defense Mechanisms*, chapter 1. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[53] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring Internet Denial-of-Service Activity. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 2–2, Berkeley, CA, USA, 2001. USENIX Association.

[54] J. Nagle. On Packet Switches with Infinite Storage. *Communications, IEEE Transactions on*, 35(4):435–438, April 1987.

[55] Bryan Parno, Dan Wendlandt, Elaine Shi, Adrian Perrig, Bruce Maggs, and Yih-Chun Hu. Portcullis: Protecting Connection Setup from Denial-of-Capability Attacks. In

Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications, SIGCOMM '07, pages 289–300, New York, NY, USA, 2007. ACM.

[56] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking (TON)*, 3:226–244, 1995.

[57] Sara Peters. CSI/FBI computer crime and security survey. Annual Report, December 2009.

[58] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[59] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168.

[60] G. Price. A general attack model on hash-based client puzzles. In *9th IMA Conference on Cryptography and Coding*, volume 2898, pages 319–331, Cirencester, UK, 2003.

[61] S. Ranjan, R. Swaminathan, M. Uysal, and E. Knightly. DDoS-Resilient Scheduling to Counter Application Layer Attacks Under Imperfect Detection. In *Proceedings of 25th IEEE International Conference on Computer Communications (INFOCOM 2006)*, pages 23–29, April 2006.

[62] Supranamaya Ranjan, Roger Karrer, and Edward W. Knightly. Wide Area Redirection of Dynamic Content by Internet Data Centers. In *Proceedings of IEEE INFOCOM*, March 2004.

[63] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, MIT, Cambridge, Massachusetts, 1996.

[64] Hosam Rowaihy, William Enck, Patrick Mcdaniel, and Thomas La Porta. Limiting sybil attacks in structured p2p networks. In *the IEEE INFOCOM '07*, pages 2596–2600, 2007.

[65] Scalable Sensing Service. Planet-lab scalable sensing service. http://networking.hpl.hp.com/s-cube/.

[66] M. Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '95, pages 231–242, New York, NY, USA, 1995. ACM.

[67] Dorgham Sisalem, Jiri Kuthan, and Sven Ehlert. Denial of service attacks targeting a SIP VoIP infrastructure: Attack scenarios and prevention mechanisms. *IEEE IEEE Networks Magazine*, 20(5), 2006.

[68] Bryan Sullivan. XML denial of service attacks and defenses. *MSDN Magazine*, November 2009.

[69] S. Tritilanunt, C. Boyd, E. Foo, and J. M. González. Toward non-parallelizable client puzzles. In *6h International Conference on Cryptology and Network Security*, pages 247–264, 2007.

[70] Limin Wang, Vivek Pai, and Larry Peterson. The effectiveness of request redirection on CDN robustness. *SIGOPS Oper. Syst. Rev.*, 36:345–360, December 2002.

[71] X. Wang and M. K. Reiter. Defending against denial-of-service attacks with puzzle auctions. In *IEEE Symposium on Security and Privacy*, pages 78–92, Washington DC, 2003.

[72] X. Wang and M. K. Reiter. Mitigating bandwidth-exhaustion attacks using congestion puzzles. In *11th ACM Conference on Computer and Communications Security*, pages 257–267, 2004.

[73] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New client puzzle outsourcing techniques for dos resistance. In *11th ACM CCS*, pages 246–256, 2004.

[74] Wikipedia. Operation Payback, November 2010. http://en.wikipedia.org/wiki/Operation_Payback.

[75] Jared Winick and Sugih Jamin. Inet-3.0: Internet topology generator. Technical Report CSE-TR-456-02, University of Michigan, 2002.

[76] Yi Xie and Shun-Zheng Yu. Monitoring the Application-Layer DDoS Attacks for Popular Websites. *IEEE/ACM Transactions on Networking*, 17:15–25, February 2009.

[77] Ying Xu and Roch Guérin. On the Robustness of Router-based Denial-of-service (DoS) Defense Systems. *SIGCOMM Comput. Commun. Rev.*, 35(3):47–60, July 2005.

[78] Che-Fu Yu and Virgil D. Gligor. A formal specification and verification method for the prevention of denial of service. *IEEE Symposium on Security and Privacy*, pages 187–202, April 1988.