Technical Report

UCAM-CL-TR-703
ISSN 1476-2986

Number 703

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Lazy Susan: dumb waiting as proof of work

Jon Crowcroft, Tim Deegan,
Christian Kreibich, Richard Mortier,
Nicholas Weaver

November 2007

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Lazy Susan: dumb waiting as proof of work

Jon Crowcroft
University of Cambridge
Computer Laboratory jon.crowcroft@cl.cam.ac.uk

Tim Deegan
XenSource tim.deegan@xensource.com

Christian Kreibich
International Computer Science Institute christian@icir.org

Richard Mortier
Microsoft Research Cambridge mort@microsoft.com

Nicholas Weaver
International Computer Science Institute nweaver@icsi.berkeley.edu

November 15, 2007

Abstract

The open nature of Internet services has been of great value to users, enabling dramatic innovation and evolution of services. However, this openness permits many abuses of open-access Internet services such as web, email, and DNS. To counteract such abuses, a number of so called *proof-of-work* schemes have been proposed. They aim to prevent or limit such abuses by demanding potential clients of the service to prove that they have carried out some amount of work before they will be served. In this paper we show that existing resource-based schemes have several problems, and instead propose *latency-based proof-of-work* as a solution. We describe centralised and distributed variants, introducing the problem class of *non-parallelisable shared secrets* in the process. We also discuss application of this technique at the network layer as a way to prevent Internet distributed denial-of-service attacks.

1 Introduction

There can be no doubt that the open nature of the Internet has great value, providing essentially unrestricted communication between hosts, enabling a host to use services openly provided by others. This openness has enabled innovation and evolution, resulting in the Internet we know today. Unfortunately today's Internet is also plagued by a daunting variety of abuses from unsolicited email to distributed network-layer denial-of-service attacks. The effects of these abuses are felt by most if not all users of the Internet and Internet-based services such as email, DNS, and the web.

The underlying cause of much of this abuse is that an attacker can cause disproportionate consumption of available resources at the hosts providing the service. In network-level distributed *denial of service* (DDoS) attacks, large numbers of attacking hosts send enough traffic to a victim to clog up its access links, consuming all inbound server network resources. Unsolicited email forms an *abuse of service* as individual nodes waste disproportionate amounts of bandwidth and CPU cycles by pumping vast amounts of email through individual SMTP servers. The goal here is not to *deny* the service to legitimate clients but to *abuse* its willingness to serve for commercial ends.

In this paper we focus on the latter problem, namely abusive use of openly available services. A dominant approach to battling this abuse, particularly in the context of unsolicited email, is to impose *proof-of-work* (PoW) [17] on the client. This forces the client to consume some of its own resources before the server will grant it service, slowing down the client's request rate and reducing the economic incentive to abuse the service.

First, we examine the dominant PoW categories – CPU-bound and memory-bound – and arrive at the conclusion that they are not suitable for all applications (§2). Second, we propose a novel avenue for proving performed work by enforcing *passive delay* on a client. We call this *latency-based* PoW (§3) and propose two different implementation strategies: centralised latency-based PoW, in which a server slows down the client through the use of *delay-cookies*; and distributed latency-based PoW, where we extend delay-cookies by requiring the client to contact a number of other hosts before being admitted to the service. As we will show, latency-based PoW requires a new kind of distributed puzzle which we term *non-parallelisable shared secrets*. Latency-based PoW has several advantages over existing approaches:

1. Independence from client resources such as CPU and power. This is an important benefit for a PoW scheme that is expected to run Internet-wide on devices as diverse as datacenter-hosted servers and hand-held mobile phones.

2. Independence from the client's other activities, since the actions of one application in proving latency-based PoW in no way reduces the ability of other applications on the same client to make progress. In particular, this avoids the problems of CPU-based PoW identified by Laurie and Clayton [19].

3. Preservation of the client's ability to establish connections to many *different* hosts in parallel. This is an important requirement of many popular present-day applications such as peer-to-peer platforms.

4. Enlargement of the client's network footprint in the distributed PoW variant, by making it visible under the same identity over a longer period of time and at more places in the network. This aids both prevention of attacks on the proof computation and post-hoc forensic analysis.

We continue by presenting an analytical comparison of resource-based and latency-based PoW (§4). We then shift focus from the transport layer to the network layer, and outline use of latency-based PoW to prevent network layer denial of service attacks (§5). Finally, we present related work (§6) and conclude with some thoughts on future work (§7).

2 Existing schemes for proof-of-work

In this section we survey the dominant classes of PoW proposed to date, and discuss their common problems as well as their suitability in different application settings.

2.1 CPU-bound work

In its original formulation, PoW was strongly tied to *computational* work, i.e., work in which the client is actively busy in order to complete the work assignment. The general idea is that the client needs to present to the server the outcome of a moderately costly computation $y = F(x)$ along with its input x. The server must then be able to verify that $y = F(x)$ does indeed hold at substantially lower cost than that invested by the client.

Dwork and Naor [12] originally termed such proofs of work *pricing functions* for the client to compute, and suggested square-root computation and cracking intentionally weakened cryptographic signature schemes as sample problems. Back [7] takes a similar approach with *HashCash*, relying on generating collisions among prefixes of hashed values.

The design space for CPU-bound PoW is large, and variants have been proposed that vary along numerous dimensions:

> **Interactive** *vs.* **non-interactive**. Does the server need to be contacted in order to obtain the work assignment?
>
> **Publicly auditable** *vs.* **private**. Can a third party verify the result?
>
> **Trapdoored** *vs.* **trapdoor-free**. Can the server leverage knowledge of *shortcuts* to generate the correct solution?
>
> **Sequential** *vs.* **parallelisable**. Is there any way to speed up the computation through parallelisation?

Notably absent from these considerations is whether the scheme is *centralised* or *distributed*; previously proposed schemes are largely centralised in that communication occurs only between the client and the server, at most including the possibility of a third party for purposes of external validation and auditing.

2.2 Memory-bound work

Another category of algorithms focusing on memory latencies rather than CPU cycles was first proposed by Abadi et al. [1, 2]. They observed that assigning universally fair amounts of CPU-bound work to arbitrary machines is hard due to the dramatic variations in computational resource available to different hosts. In contrast, memory latency exhibits much smaller variance than CPU speed. In the scheme they proposed, the server essentially chains together the computation of k applications of a function $x_{i+1} = F(x_i)$. The client's goal is to find the original x_0 by repeatedly computing $F^{-1}(x)$ which is a more costly operation than computing $F(x)$. The scheme is carefully designed to use computations that clients can best solve by first computing a table of solutions for F^{-1} over its entire domain, and then completing the work assignment using chained application of this function.

To further skew the amount of work required for constructing/solving the work assignment toward the client, F is surjective, i.e., F^{-1} can have multiple possible values, potentially forcing the client to explore multiple branches in an (implicit) tree structure. An intriguing aspect of their scheme is that it can be rendered entirely non-interactive: it is possible to architect the proof of work in such a fashion that the client can prove completed work to the server without ever having contacted either it or a third party. Abadi et.al. suggest table sizes of the order of 32MB for the client's work assignment. Dwork et al. [14] expand upon this idea, and require 16-18MB for the client's precomputed table.

2.3 Cognitive work

An entirely different class of PoW typically termed "CAPTCHAs" [30] consists of work that *cannot* practically be performed by a computer at all, while generally being trivial to solve by a human. Examples include distorted renderings of text and image recognition. The intention is typically also to prevent abuse of service usually in situations where benign use will be relatively infrequent, e.g., posting comments to blog sites or registering user accounts in web fora. These schemes have seen widespread adoption and are largely effective.[1] However, enforcement is not through brute force but through cognitive ability: CAPTCHAs cannot easily be crafted in a fashion that imposes specific computational resources since they test the absolute proposition "is the client a human or a computer?", and so we do not consider them further in this paper.

2.4 Problems with resource-based work

We believe that there are two main problems with use of resource-based (i.e., CPU- or memory-bound) PoW in modern networked applications: they expect exclusive use of the host on which they're running, and attackers can elide them through the use of large-scale botnets if the work is non-parallelisable.

2.4.1 Exclusive nature of work

Resource-bound PoW has been proposed as a solution to a wide range of problems (see Jakobsson and Juels [17] for a survey). In some sense, this is due to a benefit of resource-bound work: it obviates the need to identify malicious clients as performing the work obstructs the attacker whether or not they issue requests under a spoofed identity or their true identity. However, we argue that resource-bound PoW is flawed in the bigger picture: it assumes *exclusivity of work*, i.e., that the client can be burdened with additional work because its only goal is to use the service requiring the PoW. While perhaps true for attackers, it is not true for hosts of legitimate users which will typically be busy with many tasks.

Worse, if resource-bound PoW were widely deployed, the client might well be prevented from legitimately using *multiple services in parallel*: each piece of CPU-bound work will clearly increase the time it takes the client to complete other piece of work; with memory-bound work the client does not have the room to do the work since each

[1] Attackers have however "outsourced" such challenges to sites under their control, harnessing unknowing visitors for solving the assignments, which the attackers then forward on to the original site [13].

assignment requires substantial amounts of memory plus some CPU cycles. Since such PoW schemes actively slow down the client, attempting to complete parallel independent work assignments will increase the slowdown imposed on the client at least linearly, no matter the chosen completion strategy.

However, parallel communication with a large and diverse set of hosts is a common scenario in popular present-day applications. For example, the default settings of a peer-to-peer system such as Pastry [23] requires each node to maintain connectivity with around 16 other hosts; in a modern file-sharing system such as BitTorrent, each client maintains between 20 and 40 clients [16].

2.4.2 Large-scale parallelisation

The advent of *botnets*, large groups of compromised hosts under control of a single attacker, has caused some discussion regarding the usefulness of PoW schemes. Back [7] and Laurie and Clayton [19] point out that botnets can be used to perform massively parallel PoW, in at least three possible ways:

- A single client subdivides a parallelisable assignment and has the bots compute these subproblems, thus increasing the client's completion rate.
- A single client distributes whole assignments to the bots and has them compute them for it, thus increasing the client's completion rate.
- Each compromised host communicates with the server autonomously, parallelising completion.

We agree with this analysis: where the work in the proof can be performed faster through the action of many hosts and there exists an incentive to do so, botnets will be used in this manner. We would argue that these kind of massively parallel attacks are out of the scope of most PoW schemes, which are intended to control individual clients. Our proposal elides this problem: although latency-based PoW is parallelisable at a single client (a host can wait for many thing simultaneously), parallelisation does not lead more quickly to a solution so there is no incentive to involve a botnet. When applied at the service level this still leaves open the possibility of attacks at other layers (e.g. network layer DDoS): we outline a possible solution in Section 5.

Due to these two problems we believe that resource-based PoW is unsuitable in a general setting, and so we next propose a scheme that preserves the clients' ability to engage in parallel activity while still forcing it to slow down if so requested by its peers.

3 Latency-based proof-of-work

Stepping back for a moment we note that the basis for PoW schemes is to cause the client to do some piece of work for the server in order to partly right the imbalance in resource consumption. Currently, the client can cause the server (or multiple servers) to do large amounts of work on its behalf, while requiring almost none of its own resources to be consumed. Examples of such asymmetry of invested and effected work include sending email to a large number of recipients, uploading files into peer-to-peer sharing systems, or queries to complex search engines.

CPU-bound and memory-bound PoW schemes attempt to impose some load on the client in order to curb the excesses of attackers without significantly impacting legitimate users. Both techniques boil down to increasing the *time* an attacker must spend causing servers to do work on their behalf. As we have argued, these schemes are unsuitable in the general case, and a different approach is needed.

Consequently, we now propose *latency-based proof-of-work*, where the server occasionally causes the client to complete a time-consuming work assignment that *does not* consume any client resource before the server will do any further work on the client's behalf. The major advantage of latency-based PoW is that in contrast to resource-bound approaches, it permits the client to perform multiple concurrent assignments with little undesired impact.

At first glance this might seem to do little to protect the server, but in fact this is not the case for two reasons. First, it achieves the immediate goal of enabling the server to slow down the *aggregate* client request rate, in the same way as resource-bound PoW schemes. Second, as a beneficial side-effect, the server can now predict its future workload without maintaining per-flow state. This potentially allows it to better schedule its load, e.g., by ensuring sufficient variance in the timing of continuing traffic to prevent load spikes.

3.1 Centralised latency-based work

We now consider the design space of centralised latency-based PoW, i.e., schemes that operate exclusively between client and server, before describing a possible implementation.

3.1.1 Design space

Consider as a strawman a system in which the server responds to a client's request with a *delay(d)* message, where d is some time that the client must delay before continuing the communication. If the client does not respond after d ticks then the server will simply drop the request. If and only if the client responds after a delay of at least d ticks will the server process the client's request. Earlier requests by the client are ignored, or worse, cause the server to disconnect the client. We will later demonstrate how the latter technique can be securely implemented as a variant of SYN cookies [9] (§3.1.2).

The effect of this message is to require the client to perform some negligible amount of computation, but more importantly to *wait* for a noticeable period of time before re-contacting the server. As a result, and in a manner similar to other PoW schemes, the client must therefore entertain an ongoing relationship with the server, helping the server avoid doing work for ephemeral attackers as with current-day abuse-of-service schemes.

Ideally the server would send a *delay(d)* message every time it performed some unit of work as defined by the service, e.g., sending an email or executing a method of a remote web-service. However, this requires input from the higher layers of the service to delineate the pieces of work that the service deems significant.

While technically feasible, this would require improvements to each application that should use the delay feature. There are three approximations to this: (*i*) session-based PoW, (*ii*) connection-based PoW, and (*iii*) periodic activity-triggered PoW. In the first, completed PoW would permit the client to establish arbitrary many connections to the server for a limited period of time. In the second, PoW is required on a per-connection

basis. In the third, the delay requests are triggered automatically and periodically as part of the transport-level dialog, depending on the resource usage imposed on the server.

We argue that connection-based PoW is preferable for the following reasons. First, it is more scalable than session-based PoW as it does not require the server to track a timer for each client. Second, it enables the server to control the number of concurrent connections made by a single, possibly malicious, client. Third, since resource usage is a rather complex notion and can manifest in a variety of ways (e.g., CPU cycles, memory consumption, network bandwidth), determining correct intervals for periodic PoW is challenging. The major drawback of connection-based PoW is that applications making heavy use of pipelining over a single connection will see little benefit: the cost of the PoW is amortised over the many pieces of work carried over the single connection. In such cases, approaches (*i*) or (*iii*) should be used instead.

3.1.2 Implementation

We focus discussion here on TCP services since these are more prevalent, and are more likely to need protection at the service-level from attack. The techniques could be translated to the equivalent of a connection for UDP services, but this would be likely to require application modification. An initial suggestion for implementing centralised latency-based PoW would be to clamp the server's TCP receive window, `rwnd`. By shrinking `rwnd` to 0 bytes, e.g., with the SYN-ACK during the three-way handshake, the server can effectively stop the client from sending since any further traffic would be discarded.

While attractive and feasible in principle, this approach suffers from the major drawback of requiring the server to keep connection-level state. As a better alternative we propose *TCP delay-cookies*, an extension of SYN cookies. By slightly modifying the SYN cookie creation process and introducing a *TCP Wait Request* option, we can implement delay-cookies in the TCP stack.

The TCP Wait Request option conveys to the client how long to wait before the server will continue to handle the connection. It consists of 3 bytes: 1 byte opcode, 1 byte length, 1 byte for a positive integer representing the requested waiting time, d, in seconds, which allows for over 4 minutes of delay.

To keep TCP delay-cookies stateless, the server needs a mechanism for recording in the existing sequence number space how long a new connection should be delayed. Using the notation $x_{|i}$ to denote the low-order i bits of x and x_i to denote that x is i bits long, recall that SYN cookies consist of three components: (*i*) $T_{|5}$ where T is a server time counter incremented every 64 seconds, (*ii*) a 3-bit encoding MSS of the MSS, and (*iii*) a server-selected hash function to produce the 24-bit value $H_{24}(session, T)$ of the session parameters (client & server address, client & server port) and T. The purpose of T is to allow the server to test whether the request has timed out. For delay-cookies, instead of using the current value of T, we use T', the value of T in d seconds time. Next, we introduce an additional counter t that gets incremented every second. Finally, we modify the computation of H_{24} by shrinking the hash value to 16 bits (i.e., using H_{16}) and using the remaining 8 bits to store $(t + d)_{|8}$.

To protect the resulting 24-bit value from tampering and in order to allow the server to recover the requested delay time when receiving the response from the client, the server *encrypts* the 24 bits using a block cypher $E_{24}[K, \cdot]$, such as RC6 [21] parameterised to 24-bit block length, with a key K which the server keeps private. The resulting initial

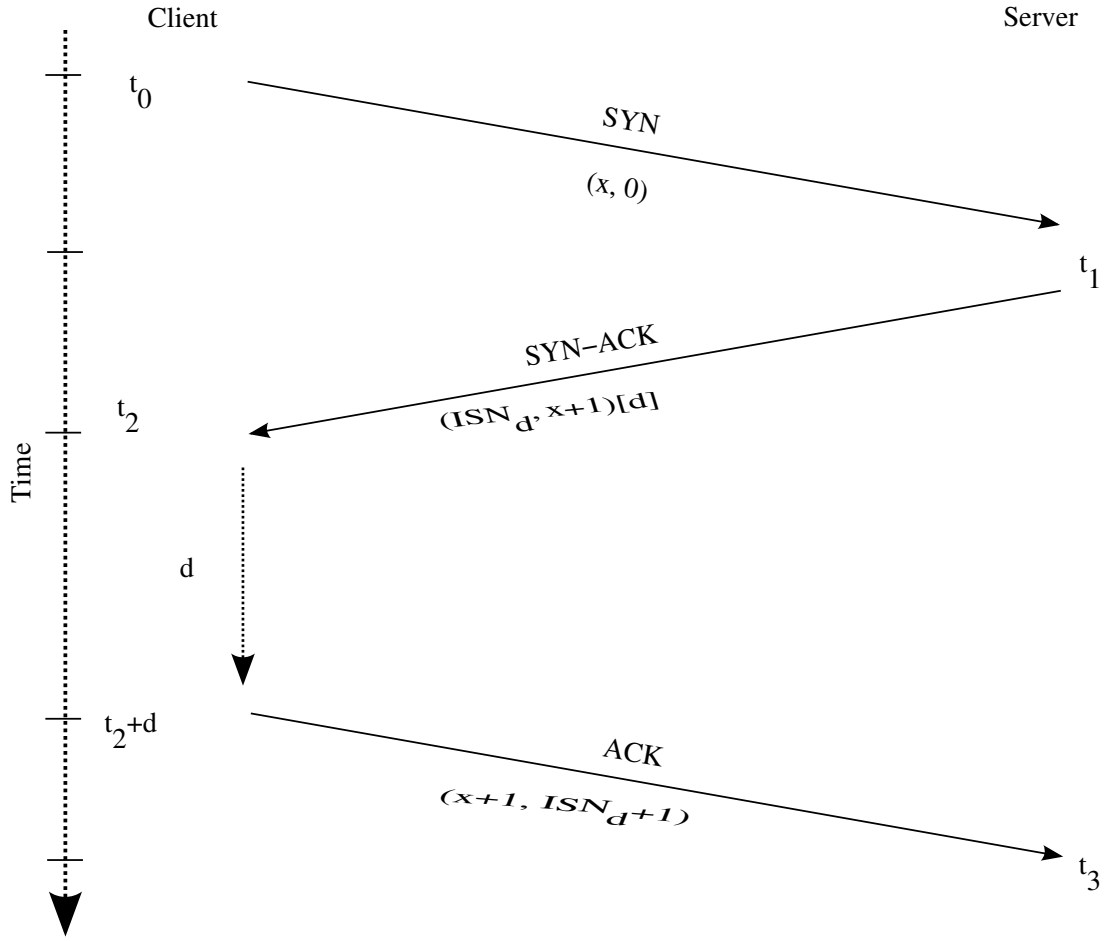

Figure 1: A delay-cookie exchange during a TCP handshake.

sequence number is thus:

$$\left(T'_{|5}, MSS, E_{24}\Big[K, \Big(H_{16}(session, T'), (t+d)_{|8}\Big)\Big]\right)$$

An example of a delay-cookie-enabled handshake is shown in Figure 1: at time t_0, the client sends a TCP SYN to request a connection. At t_1, it arrives at the server, which decides it wants to keep this client waiting for d seconds. It sends the SYN-ACK with d in a TCP Wait Request option and constructs the initial sequence number ISN_d with a timestamp of $(t_1 + d)_{|8}$ as just described. the client, who sees that it has to wait for d seconds. After waiting, it returns the ACK echoing the wait request option. time t_3 the server receives the ACK and extracts $(t_1 + d)_{|8}$ and $H_{16}(session, T')$ from ISN_d, which is the current ACK number minus one. It confirms that the hash values match and that the current time is sufficient for the requested delay as follows: It expands $(t_1 + d)_{|8}$ into t' by taking the value of t_3 and overwriting its low 8 bits. Then it checks whether $t_3 \geq t'$. Note that the hashing procedure involved T and thus ensured that the comparison occurs in the right counting cycle. If successful, the connection proceeds as normal. If too little time has passed since the SYN-ACK was sent, the server ignores the packet.

One interesting consequence is that if d is relatively small (on the order of a few seconds), TCP delay-cookies should interoperate with a normal TCP client. From the unchanged client's viewpoint, it should be interpreted as a packet loss event for d seconds

between when the SYN/ACK was received and the server decides the delay is sufficient. Unfortunately this will disrupt TCP slow-start but we do not expect this to be a common occurrence as TCP delay-cookies should only be used when a server is under attack or other stress. This should allow incremental deployment of TCP delay-cookies. Servers directly benefit from implementation, while clients will benefit by having more reliable performance when communicating with servers which are under attack.

3.2 Distributed latency-based work

We now shift our attention to an aspect of PoW that has not previously been considered in the literature to the best of our knowledge: mechanisms that non-trivially and intentionally involve multiple nodes in the proof, i.e., *distributed* PoW. Our motivation for this is twofold. First, if a node is required to engage in communication with a number of other nodes in order to prove work, this will increase its *network footprint* in terms of the spread and duration of its network presence, improving the chances of detection through coordinated investigation of abusive activity [4, 10, 27, 28]. Second, with a suitable proof scheme it enables the server to reflect the load imposed on it not only onto a single client, but on the abusive nodes as a collective entity.

3.2.1 Design space

We begin our exploration of distributed PoW schemes with a list of requirements that apply in addition to those for centralised PoW.

Non-parallelisable operation. The client must be forced to process the work in a serial fashion for the distributed aspect to make sense.

Non-centralisable operation. The client must not be able to gain advantage by centralising the processing onto itself.

Distributed secrecy. Individual nodes must not be able to help other nodes toward completing the assignment beyond their intended share of the work, to hinder collusion among the nodes involved in the proof.

Fault tolerance. The possibility of network unreliability must be anticipated.

As a first step toward these goals we build upon the idea of latency-based PoW by proposing a *distributed queue* in which the client first has to be processed by a number of nodes before being admitted to the server. This requires the client to contact several nodes in addition to the server; We call these nodes *validators*. A natural source of validators for a server are its previous clients which, from the server's perspective, conceptually form a *departure queue* preceding the server's regular request queue (see Figure 2). From the client's perspective the server points the client to a queue of validators through which access to the server can be gained (see Figure 3). This distributed scheme is thus inherently interactive. We now present a possible distributed PoW algorithm for this scenario, which essentially forces the client to obtain a shared secret as the PoW, allowing it to proceed through the distributed queue.

Shamir's classic algorithm for sharing a secret [24] would involve issuing a polynomial coordinate to each validator and requiring the client to retrieve enough coordinates in

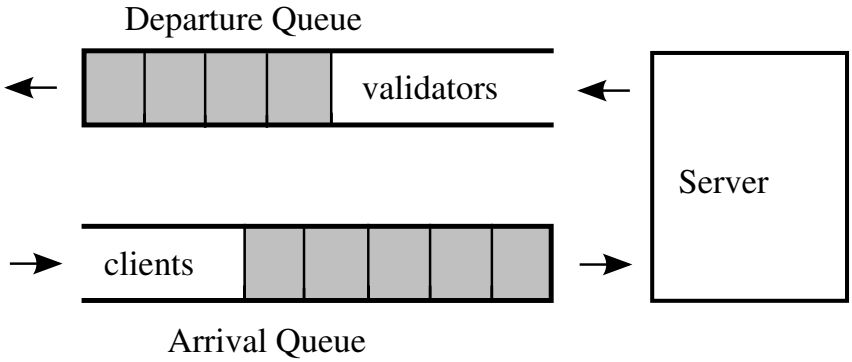

Figure 2: After being processed, the server's former clients turn into validators stored in the departure queue.

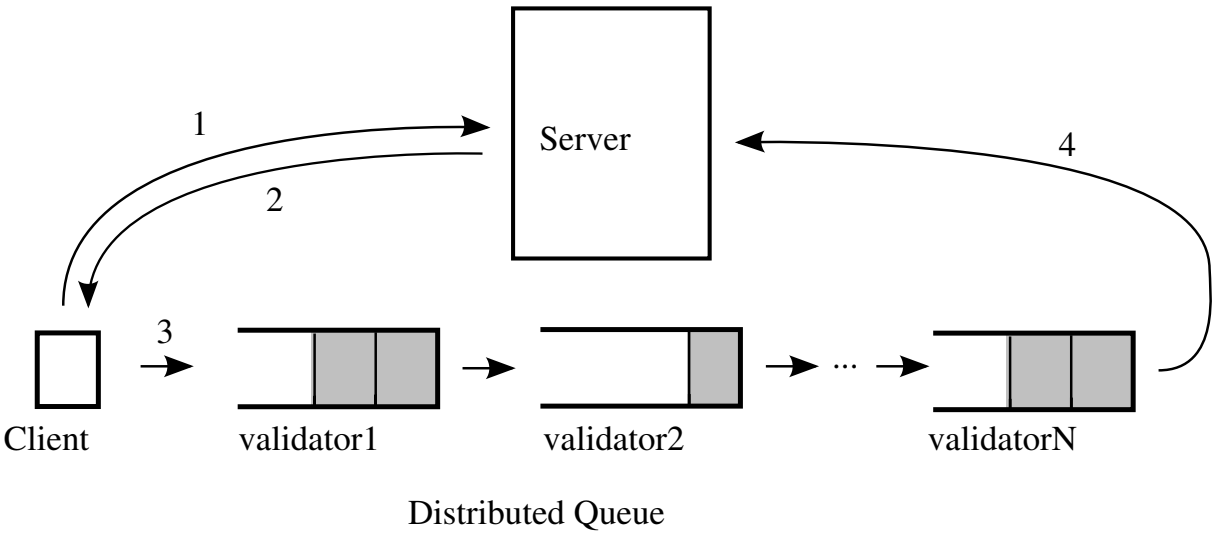

Figure 3: The client obtains details about the distributed validator queue from the server (1/2), proceeds through this queue (3), and is finally admitted to the server (4).

order to be able to interpolate the polynomial. Unfortunately, this is unsuitable in our case as retrieval of polynomial coordinate values is inherently parallelisable: the client need only issue requests to all validators in parallel to obtain the shared secret and complete the queueing process. However, Shamir's scheme does nicely provide resilience as it requires contacting only any k of n validators, each of whom knows only part of the shared secret. In contrast, the time puzzles proposed by Rivest et al. [22] are inherently resistant to parallelisation but they lack secrecy and resilience: an identical operation has to be applied a certain number of times for the puzzle to be solved.

We require an approach combining the benefits of both schemes: shared secrets, resilience, and non-parallelisability. We call this problem class *non-parallelisable shared secrets* and we now present an example of such a scheme. Note that we are not claiming it as the best or only possible approach – rather, we consider it a first step into a likely fruitful area for future research.

Let $F = \{f_1, f_2, \ldots, f_n\}$ be a set of n functions with identical input and output domains so that they are chainable. We desire that the value $y = f_{s_1} \circ f_{s_2} \circ \cdots \circ f_{s_n}$ is

a valid representation of the secret for any set $S = \{f_{s_1}, f_{s_2}, \ldots, f_{s_k}\} \subseteq F$, $s_i \in [1, n]$, of $k \leq n$ functions.

We distribute the tuple (f_i, t_i) to each validator v_i. t_i is the amount of time v_i will wait before responding to the client. We impose this delay rather than relying solely on the inherent communication latency as that can vary widely across the Internet and there is no immediate way for the server to know the latency between a potential client and the validators. Thus the server cannot estimate aggregates of such latencies accurately enough to rely on them for delay purposes. For many clients and servers it is also likely that pure reliance on communication latency would require too many verifier contacts in order to realise substantial delays.

We require the functions f_i to be *cheap* but not predictable (i.e, when $x_1 \neq x_2$, knowing that $y_1 = f_i(x_1)$ should yield no clues as to the value of $f(x_2)$), properties that hold for basic cryptographic hash functions. Then, for each validator v_i in the departure queue, the server remembers its (f_i, t_i). To force the client to wait at least T time the server sends a tuple

$$(c, S(c), k, V^n, T_{max})$$

where c is a challenge suitable for input to each of the v_i's f_i, $S(c)$ is the server's signature for c, k is the number of validators to contact, V^n is a set of n validators such that for any $V' \subset V^n$ with $|V'| = k$,

$$\sum_{v_i \in V'} t_i \geq T$$

and $T_{\max} \gg T$ is a fallback delay which the client will have to endure if it is unable to contact any $\binom{n}{k}$ validators.

The client then iteratively attempts to connect to any subset $V' = \{v_1, \ldots, v_k\} \subseteq V$ of validators. It first sends c to v_1, which computes and returns $f_1(c)$. The client then iterates, sending the most recently received result to the next selected validator, until k iterations have been made. Each individual connection is an instance of a centralised latency-bound PoW. After k verifiers have been contacted, the client returns to the server the tuple

$$(f_k(c), S(c), [v_1, \ldots, v_k])$$

where $f_k(c)$ is the result returned by the last contacted validator, $S(c)$ is as above, and the third item is a vector specifying the sequence of successful validator connections.

The server then verifies first, that c is correct by checking against $S(c)$ and second, that $f_k(c)$ is indeed the result of chained application of the validators' f_i functions in the sequence provided by the client[2]. If both are the case then the client is admitted to the service and the server may move the client from the arrival to the validator queue, possibly removing older validators. If the ex-client becomes a new validator, the server uses its current and anticipated workload to adjust the new validator's delay value t accordingly.

3.2.2 Implementation

We now consider implementation aspects of the scheme just described. As a full treatment of a protocol implementing the various exchanges would be beyond the scope of the

[2]Recall that the f_i are cheap to compute.

paper, we instead focus on a number of higher-level implementation challenges created by requiring a server's potential clients to contact its former clients.

Connectivity. The unfortunate reality is that it is not a given that any former client of a server can itself be contacted, e.g., due to the presence of substantial firewall and NAT deployments at the network edge. However, validators do not have to be the former clients of the server themselves, but could instead be agents managed by the clients' hosting organisations, reachable by definition, and made available via some similar system to reverse DNS lookup.

Privacy. Contacting former clients brings privacy issues to the fore: a malicious client might simply be interested in learning who has recently used a given service, never actually intending to make real connections. As above, this problem could be solved by having agents acting on behalf of the clients.

Bootstrap. A server's departure queue will be empty at start of day, so it has no validators at which to point arriving clients. However, this does not actually pose a problem as the server can simply issue trivially solvable assignments to initial clients. As the number of clients grows, the mechanism self-clocks since the more clients there are, the more flexibility the server has in selecting and allocating validators.

Client Identity. In addition to increasing the network footprint of the client, the scheme can be extended to force the client to use the same identity throughout the PoW. As presented, the scheme does not prevent a set of clients from communicating partially solved assignments among each other, although there is little point in doing so since each work assignment is non-parallelisable and latency-based. By parameterising the validator functions with a weak form of client identity, e.g., its IP address, the client can be forced to continue using this same identity.

4 Analysis

We now consider a simple analytical model of both resource-based and latency-based PoW schemes. We assume a system with N clients which are homogeneous in their resources for simplicity. Each client submits requests at a rate $1/T$, with a request taking the server c seconds to complete. We introduce n attackers, each submitting requests at rate $1/t$, $t \ll T$. The long-term server utilisation is therefore $Nc/T + nc/t$, and the number of clients the server can support with no attackers is $N_0 = T/c$.

Let p be the ratio of attackers to clients, i.e., $n = pN$. Then the utilisation is is:

$$\frac{Nc(t + pT)}{Tt}$$

and the proportion of the original client base that can be supported is:

$$\frac{N}{N_0} = \frac{t}{t + pT}$$

Each work assignment in the PoW scheme costs a client d ticks and is given to the client as a delay cookie (§3.1.2). As these are extensions of SYN cookies [9], the client cannot consume server resources from a false address, and the server is assumed to restrict

the number of outstanding connections to a single client address to be small number, e.g., one[3]. The number of clients that can be supported rises to:

$$N_d = \frac{(T+d)(t+d)}{c(t+pT+d+pd)}$$

Writing in terms of the original client request rate, let $r = t/T$ and $x = d/T$; then the proportion of clients supportable under attack becomes:

$$\frac{N_d}{N_0} = \frac{x^2 + (r+1)x + r}{(p+1)x + p + r}$$

and the delay required to support the full set of clients can be found by solving $x^2 + (r - p)x - p = 0$.

For example, if attackers generate requests 100 times faster than normal clients and 1% of clients are attackers, then the server should delay each client by just $T/10$ ticks to support the workload as if there were no attackers.

With resource-based PoW, a client can impose work on at most $1/d$ servers concurrently; with latency-based PoW there is no such limit as latency-based PoW parallelises[4].

Figure 4 shows impact of the rate of each attacker on the number of clients served. Even with quite a low proportion of attackers this model suggests that each attacker need not go at a very high rate before having a notable negative impact on the number of clients the server can serve. E.g., with just 1% attackers transmitting at 5× the rate of a normal client, reduces the capacity of the server by almost 40%.

Figure 5 shows the effect of the PoW enforced delay when under attack. The top two curves show the effect if 1% clients are attackers, and they communicate at 10 (top) or 100 (second) times the rate of a normal client. The bottom two curves show the same but when 10% clients are attackers. Imposing just a small extra delay of $d = T/10$ can have a dramatic effect, essentially negating the negative impact of the attackers in the top two curves.

5 Network-layer proof-of-work

Thus far we have presented latency-based PoW as a service layer technique: a way to allow servers to dissuade attackers by increasing the cost of having the server perform work on a client's behalf. However, a perennial problem with systems such as the service-layer one we have proposed is that, even if they are secure in themselves, it is typically possible to attack them at a lower layer. A concrete example of this is found in the Internet: if the victim service is itself secured, a distributed denial-of-service attack at the network layer can overload access links to the servers hosting the service, effectively closing the service down. Again, this is an example of the exploitation of the open any-to-any communication nature of the Internet. As a result we now discuss implementation of latency-based PoW as a *network layer* mechanism.

[3]This is not a problem for pipelining protocols such as HTTP/1.1; for legacy protocols such as HTTP/1.0, a very small number of connections, e.g., four, could be permitted rather than just one.

[4]In practise there would be a limit due to fixed resources at the client such as memory footprint, socket availability, etc. These resources are not relevant to this discussion.

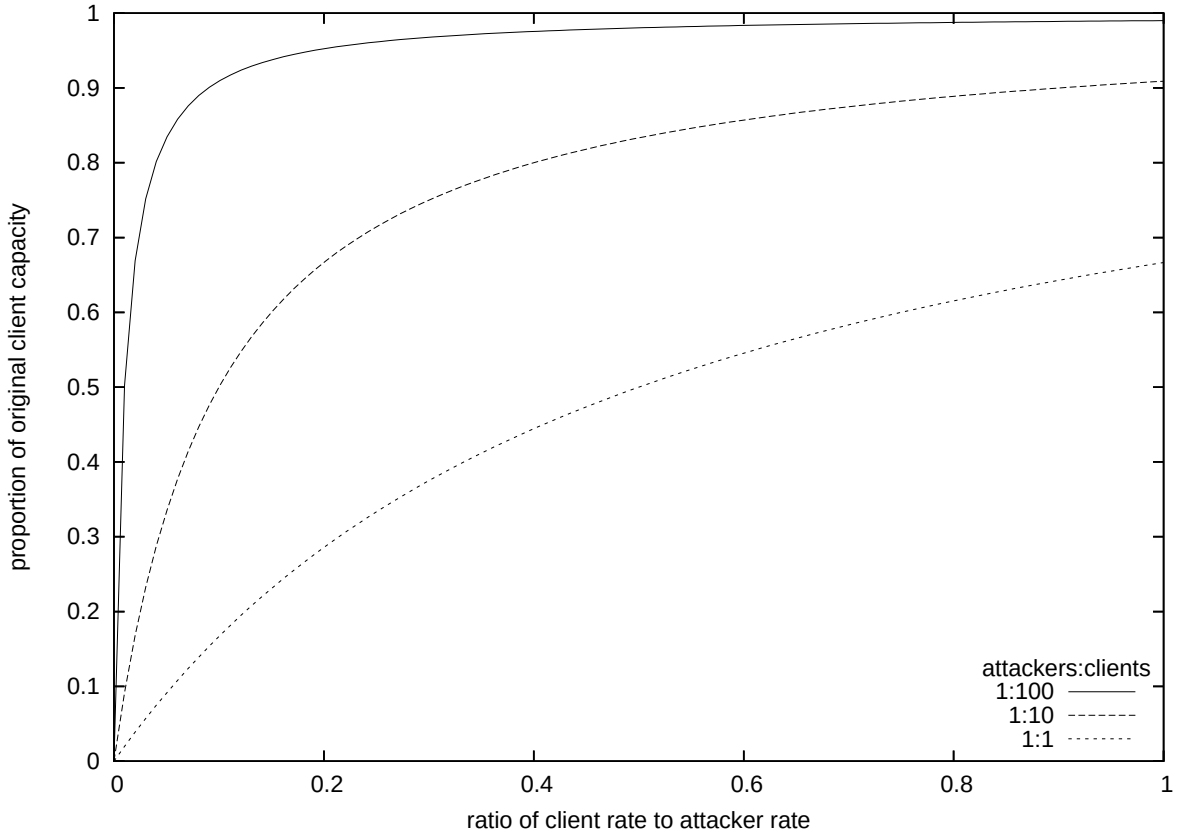

Figure 4: Effect of attacker rate on number of clients served.

Prevention of this kind of abuse requires only two simple mechanisms to extend the current Internet architecture. The first is a distributed latency-based PoW system as presented above (§3.2). The second is the ability to insert firewall rules at a network's gateways, whether these be the CPE, the first-hop router, or even the border router between two networks. Although not immediately available, similar facility is the subject of several standards [25, 26][5].

We now sketch a design for such a system, assuming for simplicity of exposition that the first-hop gateway of every client is beyond the client's reach, and is able to support rate-limiting and firewalling of traffic from and to particular hosts. Logically speaking, the address space is divided into 3 parts which we label "access" (A), "validation" (V), and "privileged" (P). In IPv4 the total address space is small enough that this logical separation cannot be matched by a physical separation but instead is implemented by mutating a set of firewall rules in the gateway. If the total address space was large enough to support this logical separation then a less complex implementation might have each router applying rate limits uniformly to the different address spaces.

The current Internet assumption of unrestricted communication now holds only for packets with the destination address from P-space. All packets for addresses in A- and V-space are subject to a rate limit which substantially restricts traffic to these spaces, in analogy with approaches taken by capability schemes that attempt to prevent flooding

[5]See IETF `draft-eggert-middlebox-control-survey-00` for a survey in-progress as of 23/June/2007.

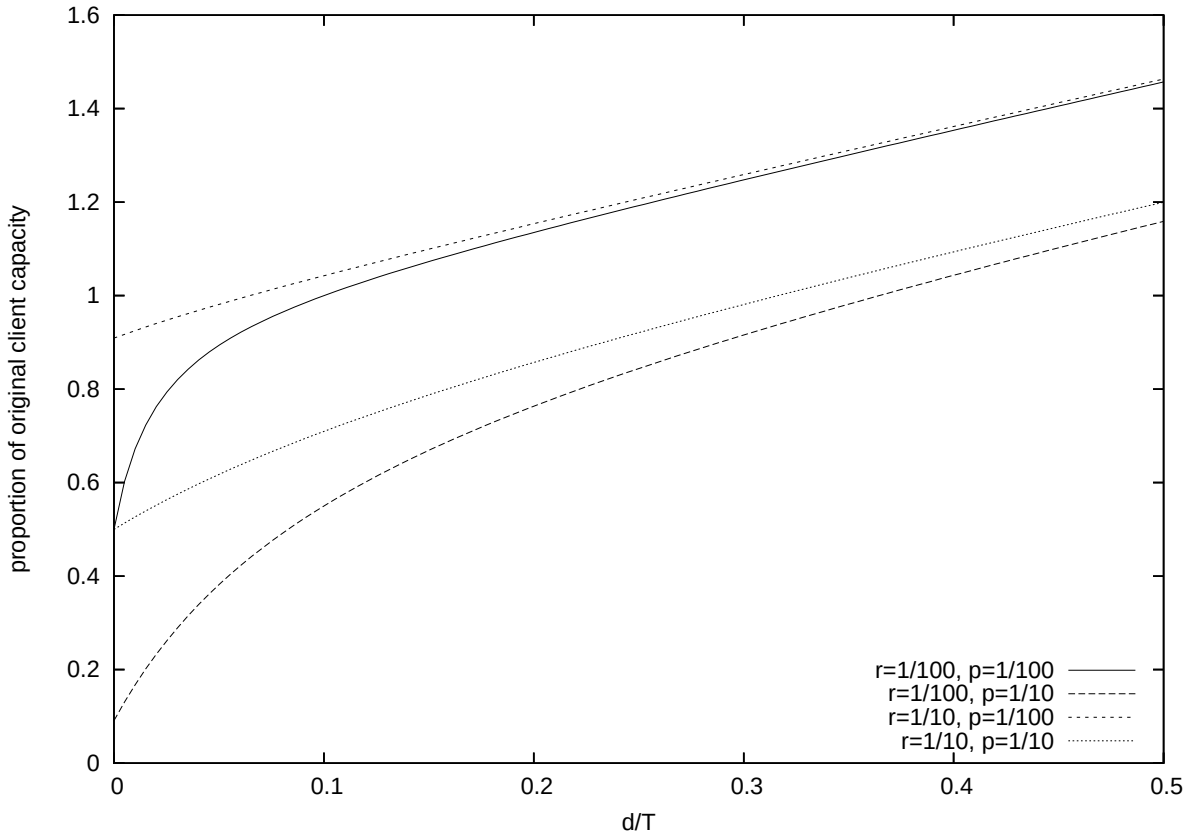

Figure 5: Effect of PoW-enforced delay when under attack.

attacks on the capability-issuing channel [5].

Once the client has recovered the S_P it uses that for further communication with S. To prevent snooping of P-addresses, they are periodically re-issued, first to validators and then to existing clients, ensuring a smooth hand-over. All traffic for A addresses is rate-limited, preventing DDoS attacks to this space. Communication with validators is similarly rate-limited, albeit to a rate k times higher than the rate to A-space as k validators must be contacted. Traffic to P-addresses is not rate limited at all once the server has confirmed admittance of the client.

A strawman scheme for implementing this in a cooperating gateway firewall/router is as follows. Let $S_X, X \in (A, V, P)$ be the server's addresses, where the client initially communicates with S_A. Observing a new flow, the gateway rate-limits it to $1/T$. The server responds to the client's request with a list of validators, $v_1 \ldots v_n$, which the gateway snoops. The gateway then installs rate-limits of k/T for all pairs (C, V_i), allowing the client to recover the necessary k pieces of S_P.[6] Finally, the client initiates communication with S_P, whose first response packet contains a special IP option to indicate that S accepts communication from C to S_P unreservedly, at which point the gateway can remove all rate-limits between that address pair.

We now provide some analysis of the resilience of this system against several variants

[6]The scheme proposed in Section 3.2 can produce a large number of different results making S_P unlikely to be unique. One workaround is to consider the results as "unlock codes" for the final S_P address which would be revealed by the server's next response.

of an attack by a set of coordinated hosts or botnet. We begin with a set of N clients, $C = \{c_1 \dots c_N\}$, and assume all communications are homogeneous in their transmission rates R, latencies t, and message loss probabilities p. Clients may speak to other hosts at rate $1/T \ll R$ unless they are speaking to a host via its P-address when they may use their full rate, R. When a client wishes to recover the address S_P, it is directed by S to speak consecutively to k of n validators, $V = \{v_1 \dots v_n\}$. Thus, to recover S_P, a client might expect to communicate with $k/(1-p)$ validators on average. P-addresses are recycled periodically, with period T.

We consider three possible attacks that may be carried out by a botnet B composed of many hosts: (*i*) attacks against a particular client, C_v, rendering it incapable of communication; (*ii*) attacks against the P-address of a particular server, S_P; (*iii*) attack a sufficiently large subset of V such that no client can recover S_P.

1. Attacking a particular client requires $|B| = T \cdot R$ since each member of the botnet can only transmit to the victim at rate $1/T$, and the victim can handle traffic up to rate R.

2. Attacking the P address of a server only requires $|B| = 1$ since the attacker must first have recovered the P address which allows it to transmit at full rate, R. *However*, it can easily be closed down by removal of the firewall entry for (B, C_v). Further, the periodic recycling of P-addresses means that B must reacquire S_P every T ticks, implying that useful work may be performed by other clients of S_P for

$$\frac{k \cdot t}{(1-p) \cdot T} \text{ ticks.}$$

3. To attack sufficient validators to prevent recovery of S_P, the attacker might proceed in one of two ways: (*i*) request V from S_A and then target $n - k + 1$ members of V; or (*ii*) construct a botnet large enough that it can attack at random enough hosts, $H' \subset H$, that S cannot select a large enough k and n.

   In the first case,

$$|B| = \frac{T}{k} \cdot (n - k + 1) \cdot R$$

   assuming that V is freshly constructed every T ticks and communication with members of V is at rate k/T.

   In the second case, the attacker must attack each host with a botnet B_i of size $|B_i| = T \cdot R$. Our scheme requires $n > k/(1-p)$ lest we run out of validators, which implies $p < 1 - k/n$. If we assume that failure to communicate with a validator is entirely due to the attack, then $p = |H'|/N$ which means that $|H'| > N \cdot (1 - k/n)$ is the required number of botnets of size $T \cdot R$ to cause the system to fail, i.e., to break the system, the attacker must control a botnet of size

$$|B| = N \cdot T \cdot R \cdot (1 - k/n)$$

6 Related work

Existing techniques for proving the completion of a given amount of work have already been discussed at length in Section 2. The first use of CPU-bound puzzles was proposed by Merkle [20] in the context of an early variant of public-key cryptography. Many other suggestions arose in the context of unsolicited electronic mail, and have typically focused on work that is CPU-bound [18, 17, 7]. Laurie and Clayton have pointed out the need for this work to be non-parallelisable [19]. The problem of the amount of work being hard to specify due to the huge variation in CPU speeds led to the proposal of memory-bound PoW [1, 2] as memory latencies vary considerably less than CPU speeds.

DomainKeys [3] provides for email authentication: rather than prevent abuse, it enables abuse to be detected and tracked more easily by having the initial mail-server sign the mail so that the receiver can verify that the mail did indeed come from the server it claims to.

Greylisting[7] can be viewed as an email specific application of latency-based PoW, relying on the ability of SMTP to temporarily fail a mail delivery attempt. The underlying aim is to delay spam to the extent that other filters can catch up and drop the spam completely. The mail server initially blacklists all mail with a previously unseen triplet *(upstream host IP, envelope sender address, envelope recipient address)*. Most spambots will not retry whereas genuine mail will, succeeding after the greylisting server decides to whitelist mail with that triplet. The authors suggest a default delay period of 1 hour although they do note that a delay of 1 minute would block 99% of the mail that the longer delay would block. Triplets remain on the whitelist for a period of time, by default 36 days, chosen as a reasonable tradeoff in the size of the whitelist against unnecessarily delaying mail. In their whitepaper the authors present results from testing which suggest Greylisting is quite successful in reducing spam, and anecdotal evidence seems to support this. However, it is specific to the SMTP protocol and implementation is somewhat tricky due to the vagaries of various mail transport agents.

Other approaches to achieving "fair" usage of distributed resources have been tried, notably in peer-to-peer filesharing systems. For example, Samsara [11] forces each node that uses resources of some other node to store a *claim* on behalf of that other node. The node storing data tests the claim from time-to-time, and if the test fails it probabilistically drops the stored data. The result is that a cheating node will eventually suffer data loss, enforcing symmetry in resource usage: a node must be willing to contribute something to others in the system.

SHARP [15] provides for a richer set of resources to be *traded* among systems, using resource claims which are unforgeable and assert control over resources. Claims are initially "soft" (*tickets*), i.e., the do not guarantee resource usage, but become "hard" (*leases*) with the agreement of the site manager, and ticket holders may delegate resources (asymmetrically) to others. Tickets have similarities to capabilities but they refer to potential future resource usage and they may expire. Detecting whether a site is actually providing the resources it has contracted out is left to some other mechanism.

Rather than trading explicit current and future usage of real resources, some systems rely on maintaining some measure of the *reputation* of each node over time. This permits local asymmetries in resource provision and consumption while ensuring that malicious

[7] `http://greylisting.org/articles/whitepaper.shtml`, revised August 21, 2003.

nodes cannot consume too much without providing some in return. For example, KARMA [29] uses certain nodes to securely track the resource provision and consumption of other nodes in the system—their *karma*. Nodes that perform the tracking are rewarded for doing so, and anti-inflation/-deflation mechanisms are provided that ensure that the price for providing/consuming resources does not under-/over-flow. Nodes without sufficient karma (free-riders) are simply not allowed to perform operations that would consume the resources of other nodes within the system.

Distributed latency-based PoW might also be seen as a distributed version of port-knocking. Port-knocking[8] [8] is a technique to obscure which ports expose services, while still granting friendly clients access to public-facing services. The client sends a sequence of SYNs to different port numbers at the server in such a way that the server can verify the client is validly "knocking" and opens the service as a result.

Capability-based strategies for prevention of DoS attacks have received considerable attention. Among the first such strategy was that of Anderson et al. [5], overlaying a network of capability servers and validators on the Internet. Capabilities are created through chained application of a cryptographic hash function usable by the clients up to packet-batch granularity, once established along the path. SIFF [31] extended this approach by eliminating the need for per-flow state in routers, instead making the routers transparently add and verify capability-suitable information of passing packets. Yang et al. [32] introduced TVA, a traffic validation architecture that further builds upon the above work. Their design adds fine-grained volume control to the capabilities and flexible destination policies, but requires all routers in the network to verify capabilities that are an additional feature of each packet. All these proposed solutions are generally more heavyweight and low-level than latency-based PoW, requiring major changes to the infrastructure.

More recently Argyraki and Cheriton [6] have posited that the notion of a protected capability channel obviates the need for a capability system, because the protection system could then simply be used for all traffic. Essentially we argue for precisely this mindset, but we apply it per-request at the application layer rather than per-packet at the network layer.

7 Conclusion

Internet communication has historically been "any-to-any", and although this openness has been crucial in encouraging the Internet to grow and develop, it does nothing to help prevent malicious usage of the network. In particular, it is relatively straightforward for clients to attack services simply by overloading them with work since modern-day clients connected over broadband access networks can impose work on a server with negligible cost.

In this paper we have discussed the shortcomings of existing so-called *proof-of-work* schemes, which aim to alleviate this problem by increasing the cost to the client of imposing work on the server. Any such scheme aims to cost the attacker either time or money, and in the case of existing schemes, both. However, existing resource-based schemes do neither well: the wide variation in the resources of Internet-connected hosts makes it difficult to precisely control the time the attacker must spend; yet CPU and memory are both too inexpensive to cost the attacker enough money [19].

[8] `http://www.portknocking.org/`, revised May 1, 2006

Thus, rather than attempt to match CPU- or memory-bound work to time, we proposed a new PoW technique that costs the attacker only time. It is both more effective at doing so, while also being simpler to implement and less burdensome on clients. *Latency-based PoW* uses the simple idea of enabling the server to delay the client between its initial (transport layer) contact and the client's imposition of work on the server. We extended this to a *distributed* latency-based PoW scheme, involving many hosts in the proof of work, thus increasing the network footprint of any attacker as an aid to forensics. In the process, we introduced *non-parallelisable shared secrets*, a novel secret-sharing strategy that combines aspects of classic approaches to time-puzzles and shared secrets. Finally, we outlined a way for such a mechanism to be applied at the network layer as a way to prevent network-layer denial of service attacks, which currently can only be dealt with at significant cost either through manual intervention (imposition of filters, route blackholing, etc) or over-provisioning of capacity.

In future work we will expand on the network layer application of distributed latency-based PoW, and also investigate other techniques enabling limited parallelisation of PoW at the client, e.g., for support of PoW in peer-to-peer systems. One possibility we are considering here is controlling the size of the table required by memory-based PoW in relation to the cache size so that a small number of proofs can be efficiently carried out simultaneously but too many causes the cache to be too small.

8 Acknowledgements

We are indebted to Ant Rowstron at MSR and members of the SRG in Cambridge for comments on the scheme.

References

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. *Proc. 10th Annual Network and Distributed System Security Symposium (NDSS)*, 2003.

[2] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. *ACM Transactions on Internet Technology (TOIT)*, 5(2):299–327, 2005.

[3] E. Allman, J. Callas, M. Delany, M. Libbey, J. Fenton, and M. Thomas. DomainKeys Identified Mail (DKIM) Signatures. RFC 4871, IETF, May 2007.

[4] M. Allman, E. Blanton, V. Paxson, and S. Shenker. Fighting coordinated attackers with cross-organizational information sharing. In *Proc. Fifth Workshop on Hot Topics in Networks (HotNets-V)*, November 2005.

[5] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet denial-of-service with capabilities. In *Proc. Second Workshop on Hot Topics in Networks (Hotnets-II)*, Cambridge, Massachusetts, USA, November 2003.

[6] K. Argyraki and D.R. Cheriton. Network capabilities: The good, the bad, and the ugly. In *Proc. Fourth Workshop on Hot Topics in Networks (HotNets-IV)*, College Park, Maryland, USA, November 2005.

[7] A. Back. Hashcash – a denial of service counter-measure. `http://www.hashcash.org/papers/hashcash.pdf`, August 2002.

[8] P. Barham, S. Hand, R. Isaacs, P. Jardetzky, R. Mortier, and T. Roscoe. Techniques for lightweight concealment and authentication in IP networks. Technical Report IRB-TR-02-009, Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA 94704, July 2002.

[9] D.J. Bernstein. SYN cookies. `http://cr.yp.to/syncookies.html`, 1997.

[10] D. Burroughs, L. Wilson, and G. Cybenko. Analysis of distributed intrusion detection systems using bayesian methods. In *Proceedings of IEEE International Performance Computing and Communication Conference*, April 2002.

[11] L.P. Cox and B.D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *Proc. ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 120–132, October 2003.

[12] D. Cynthia and P. Naor. Pricing via processing or combatting junk mail. *CRYPTO*, 92:139–147.

[13] C. Doctorow. Solving and creating CAPTCHAs. `http://www.boingboing.net/2004/01/27/solving_and_creating.html`, January 2004.

[14] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. Springer Verlag, August 2003.

[15] Yun Fu, J.S. Chase, B.N. Chun, S. Schwab, and A. Vahdat. SHARP: an architecture for secure resource peering. pages 133–148, 2003.

[16] M. Izal, G. Urvoy-Keller, E.W. Biersack, P.A. Felber, A. Al Hamra, and L. Garcés-Erice. Dissecting BitTorrent: Five months in a torrent's lifetime. In *Proceedings of the 5th anuual Passive & Active Measurement Workshop (PAM 2004)*, Antibes Juan-les-Pins, France, April 2004.

[17] M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. pages 258–272. Kluwer, BV Deventer, The Netherlands, 1999.

[18] A. Juels and J. Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. *Proc. Network and Distributed System Security Symposium (NDSS)*, 99:151–165, 1999.

[19] B. Laurie and R. Clayton. 'Proof-of-Work' proves not to work. In *Proc. Third Annual Workshop on Economics of Information Security (WEIS)*, Minnesota, USA, May 2004.

[20] R.C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978.

[21] R.L. Rivest, M.J.B. Robshaw, R. Sidney, and Y. Lisa Yin. RC6 block cypher. `http://www.rsa.com/rsalabs/node.asp?id=2512`.

[22] R.L. Rivest, A. Shamir, and D.A. Wagner. Time-puzzles and time-release crypto. `http://people.csail.mit.edu/rivest/RivestShamirWagner-timelock.pdf`, 1996.

[23] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[24] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[25] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan. Middlebox communication architecture and framework. RFC 3303, IETF, August 2002.

[26] M. Stiemerling, J. Quittek, and C. Cadar. NEC's Simple Middlebox Configuration (SIMCO) Protocol Version 3.0. RFC 4540, IETF, May 2006.

[27] J. Stutzman. Introduction to triangulation in attack analysis (Part I). *The Internet Security Conference Newsletter (TISC) Insight*, 3(6), March 23 2001.

[28] J. Stutzman and D. Lemmon. Introduction to triangulation in attack analysis (Part II). *The Internet Security Conference Newsletter (TISC) Insight*, 3(10), May 18 2001.

[29] V. Vishnumurthy, S. Chandrakumar, and E.G. Sirer. KARMA: A secure economic framework for P2P resource sharing. In *Proceedings of the Workshop on the Economics of Peer-to-Peer Systems*, Berkeley, California, June 2003.

[30] L. von Ahn, M. Blum, N.J. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. *Lecture notes in computer science*, pages 294–311.

[31] A. Yaar, A. Perrig, and D. Song. SIFF: a stateless Internet flow filter to mitigate DDoS flooding attacks. In *Proc. IEEE Symposium on Security and Privacy*, pages 130–143, 2004.

[32] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. *Proc. Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 241–252, 2005.