mod_kaPoW: Protecting the Web with Transparent Proof-of-Work

Ed Kaiser
Portland State University
edkaiser@cs.pdx.edu

Wu-chang Feng
Portland State University
wuchang@cs.pdx.edu

***Abstract*— Attacks from automated web clients are a significant problem on the Internet. Web sites often employ Turing tests known as CAPTCHAs to combat automated agents. Unfortunately, such defenses require frequent human user input, are becoming less effective as computer vision techniques improve, and can be subverted by adversaries willing to hire humans to solve challenges.**

Several alternative defenses based upon cryptographic methods rather than human input have been proposed to achieve the same goals. Such "proof-of-work" techniques prioritize clients based on their willingness to solve computational challenges of client-specific difficulty set by the server. Unfortunately, few proof-of-work schemes have been deployed since they require wide-scale adoption of special client software to operate properly.

To address these problems we present `mod_kaPoW`, a novel system that has the efficiency and *human-transparency* of proof-of-work schemes as well as the *software backwards-compatibility* of CAPTCHA schemes. The system leverages common web technologies to deliver a challenge, solve it, and submit the client response, while providing accessibility for legacy clients. This paper describes and evaluates a prototype of this system.

I. Introduction

Attacks from automated web clients are a significant problem on the Internet. Such attacks include comment spam on web-based forums [1], ticket-purchasing robots [2], click-fraud robots, and denial-of-service attacks. One way to combat this problem is the use of image-based CAPTCHAs [3]. A CAPTCHA is an automated Turing test typically consisting of skewed representations of letters and numbers, which a user must correctly interpret before they are given access.

There are several disadvantages of using CAPTCHAs. One drawback is the user-interface problem they create [4]; users with visual disabilities are unable to access content legitimately while normal users find it increasingly difficult to solve CAPTCHAs correctly as the images have become less readable in order to thwart sophisticated adversaries that have developed automated solvers for simple CAPTCHAs [5]. Another drawback is the static nature of the problems being given out. CAPTCHAs are designed to produce an image that a user can solve on the order of tens of seconds. Enterprising adversaries have outsourced the solving of CAPTCHAs with rates as low as $3 for every 1000 images solved [6]. From an economic standpoint, the inability to change the "price" of access threatens the utility of the system [7].

An alternative to CAPTCHAs is the use of proof-of-work (PoW) protocols. A proof-of-work scheme alters the operation of a network protocol so that a client must return their challenge along with a correct answer before being granted service. The challenge acts as a filter for clients based on their willingness to solve a computational task of varying difficulty. The difficulty is tailored to the individual client and is proportional to its load on the server. There have been several proof-of-work systems proposed in the literature [8], [9], [10], [11], [12], [13]. While such systems are highly configurable in terms of workload, few have actually made much progress towards being deployed. The biggest problem with these schemes is that they require changes to standard protocols and wide-scale adoption of special client software in order to operate properly, denying all clients who have not installed it.

To address this problem, this paper describes the design, implementation, and evaluation of a novel web-based proof-of-work system that provides the benefit of configurable PoW protocols in a deployable manner. Unlike CAPTCHAs, the system is *transparent* to its users and supports *backwards compatibility* for legacy clients. The basic approach only requires changes to web servers and is similar to the URL rewriting approach employed by content-distribution networks such as Akamai. In the approach, the web server dynamically rewrites URL references by attaching a computational puzzle to them. It also sends with these references, a small piece of JavaScript code for solving the puzzle embedded in the references. Valid URLs are dynamically calculated by invoking the JavaScript solver.

978-1-4244-2219-7/08/$25.00 (c)2008 IEEE

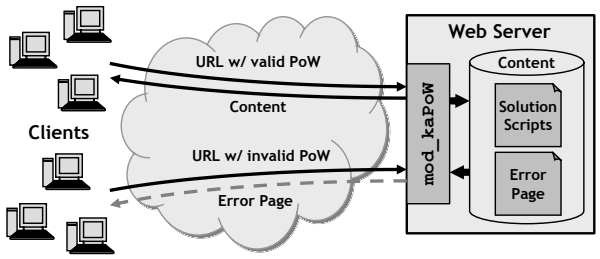


Fig. 1. The novel proof-of-work system, highlighting its additions to typical web service.

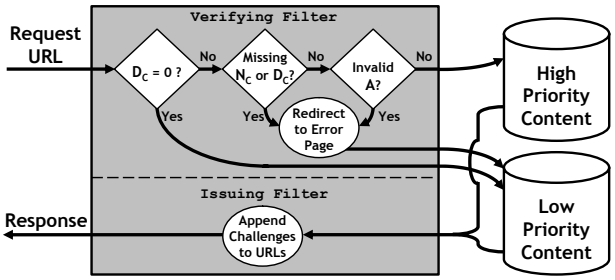


Fig. 2. The `mod_kaPoW` Apache module showing the processing of a URL and the corresponding content.

II. System Design

A. Overview

The novel proof-of-work system we call `mod_kaPoW` is shown in Figure 1. The system protects web content by embedding challenges (called *work functions*) and solutions in Uniform Resource Locators (URLs) as new query parameters. As webpages are served, the URLs found in any Hyper-Text Markup Language (HTML) tags are updated to include a proof-of-work challenge.

The system leverages the pervasiveness of JavaScript, software present and enabled on upwards of 94% of modern web browsers [14]. When a client's browser finds a PoW-protected link, it runs a server-provided script to solve the challenge and append the solution to the URL. As elaborated upon later, clients that do not have JavaScript enabled are not necessarily prevented from accessing the content.

Upon receiving a request, the server verifies that the URL contains a valid challenge and correct solution before servicing it. If either the challenge is stale or the solution is incorrect, the system denies the request and returns an error page containing a link to the resource and a new challenge.

B. The Work Function

While this system could use any of several different types of work functions, the prototype uses the compact Targeted Hash-Reversal function [9] of the form:

$$H(N_c \mid\mid D_c \mid\mid A) \equiv 0 \bmod D_c \qquad (1)$$

where H is a pre-image resistant hash function with output uniformly-distributed, N_c is a client-specific nonce generated by the web server, D_c is the client-specific difficulty set by the web server, and A is the solution that the client's JavaScript solver must find. Since both N_c and D_c are fixed by the issuer and H is pre-image resistant, this work function requires the solver to try various values for A until the equation is satisfied. This is expected to take D_c tries on average because the output is uniformly-distributed.

This work function is a good candidate because it is efficiently implementable. Specifically, using the SHA-1 hash function, it can be verified in a single hash, taking $1.09\mu s$ in software [8]. Further, the work function can be expressed compactly; the issuer simply has to send D_c and N_c, and a verifier only needs those parameters and the solution A to verify that Equation 1 is satisfied.

C. The Server Module

The bulk of the system lies within an Apache web-server module shown in Figure 2. Apache provides a rich interface for writing modules that range from those that control how a client accesses a server (such as `mod_rewrite`) to those that dynamically generate content (such as `mod_include`). As a result, Apache lends itself well to supporting a proof-of-work module[1]. The module has two filters; an *issuing filter* that embeds proof-of-work challenges in outbound content and a *verifying filter* that prioritizes inbound requests based on having a valid challenge with a correct answer.

1) Server Configuration: To prioritize requests, the server is configured with two virtual hosts. The default low-priority virtual host does not support persistent HTTP connections, and tears down any connection after servicing a single request. Only a limited number of low-priority clients are handled concurrently; all additional low-priority clients are rejected with error code `503: Service Temporarily Unavailable`.

A request demonstrating a correct answer to a valid challenge is redirected to the high-priority virtual host, which supports more concurrent clients and does not tear down connections. However, once a high-priority client sends an invalid request, the connection is redirected back to the low-priority virtual host to be terminated.

[1]The Apache module naming-convention inspired the name "`mod_kaPoW`".

978-1-4244-2219-7/08/$25.00 (c)2008 IEEE

2) Client-Specific Variables: During the course of their operation, both filters refer to the client-specific difficulty D_c and nonce N_c.

To establish D_c, the module uses a counting Bloom filter [15], [16] to track the load imposed by individual clients. The counting Bloom filter is an efficient data structure that offers a tradeoff between size and the probability of incorrectly assigning a high difficulty to a client. It has no false negatives (i.e. a client known to be malicious will never be issued a trivial work function), and the probability of a false positive can be driven arbitrarily low with additional memory. Given that the Bloom filter uses k different hash functions to index into an array of n counters, the probability of misclassifying a single client from an estimated population of m clients is approximately $(1-e^{-\frac{km}{n}})^k$. Using a value of k that minimizes that equation, the error is approximated by $0.6185^{\frac{n}{m}}$. Thus to achieve a misclassification rate of less than 0.1% of 20,000 clients, the Bloom filter requires 288,000 counters or a total of 1.2MB when using 32-bit counters. The Bloom filter is updated in every 10 seconds so that the difficulty is held constant long enough to give clients a chance to respond but short enough so that the difficulty can respond to sudden changes in load. When the structure is updated, each counter c is updated according to the following logic:

$$c_{t+1} = \begin{cases} c_t + requests_t - decay & requests_t \leq decay \\ c_t + 1.01^{requests_t - decay} & \text{otherwise} \end{cases} \quad (2)$$

which states that the difficulty decays linearly from one time window to the next unless the *requests* in the last time period t are greater than the rate of *decay*, in which case those extra requests count exponentially towards increasing the difficulty.

The client-specific nonce N_c is created by concatenating the client's identity IP_c, the unmodified URL, and a server nonce N_s;

$$N_c = IP_c \;||\; URL \;||\; N_s \quad (3)$$

binding the nonce and entire work function, to the client and specific content for a fixed window of time. When the server nonce changes the existing client nonces effectively expire, meaning solutions cannot be reused indefinitely. The unpredictable server nonce prevents the offline solving attacks that have been employed against CAPTCHAs [6]. The server can update its nonce independently from the Bloom filter, as frequently as needed to keep client solutions fresh, however the prototype updates the nonce and Bloom filter simultaneously.

```
<HTML>
  <HEAD>
    <SCRIPT TYPE='text/javascript' SRC='kaPoW.js'></SCRIPT>
    <TITLE>Sample Content Page</TITLE>
  </HEAD>
  <BODY>
    <H1>Sample Content Page</H1>
    This webpage demonstrates an image and link protected by
    proof-of-work.<BR><BR>
    <IMG SRC='test.jpg?Dc=0' Nc=a53b6145 Dc=10> are solved when
    the page is loaded to avoid delay.<BR>  In contrast,
    <A HREF='/?Dc=0' Nc=52a6c561 Dc=10>
    PoW-protected links</A> are solved only when the link is clicked.
  </BODY>
</HTML>
```

Fig. 3. The HTML markup of a sample document highlighting a link to the solution script file and the work function variables that were added.

```
<HTML>
  <HEAD>
    <SCRIPT TYPE='text/javascript' SRC='kaPoW.js'></SCRIPT>
    <TITLE>Error: Invalid PoW</TITLE>
  </HEAD>
  <BODY ONLOAD='Solve(document.links[0]);
                window.location.replace(document.links[0].href)'>
    <H1>Invalid PoW</H1>
    The requested URL did not have a valid proof-of-work attached.<BR>
    If you are reading this page, it is likely that you do not have
    JavaScript enabled.<BR><BR> If you would still like to try to
    access the content, please click the following link:
    <A HREF='http://maes.cs.pdx.edu/?Dc=0' Nc=52a6c561 Dc=10>
    http://maes.cs.pdx.edu/</A><BR><BR><HR>
  </BODY>
</HTML>
```

Fig. 4. The HTML markup of an error page sent in response to a URL that had an invalid or missing solution. The browser refresh script is highlighted.

3) The Issuing Filter: The issuing filter scans and parses HTML documents as they are served. It adds work functions to **all** tags containing URLs, as well as the instructions necessary for a client's browser to solve the functions as shown in Figure 3.

The issuing filter includes the solution instructions for work functions through the addition of a link to a JavaScript file (*kaPoW.js*) at the head of the document so that it is retrieved first (unless already cached) and the script may work as the remaining tags are incorporated into the client's in-memory Document Object Model (DOM). Despite containing a URL, this tag does not have a work function because clients need this resource before they can possibly solve any work function.

The issuing filter incorporates work functions into tags by adding the variables N_c and D_c as tag attributes. To avoid accidentally triggering HTML escape sequences, the values are transmitted in hexadecimal. It is important to observe that N_c differs between tags because it is calculated from the original unmodified URL of each tag (recall Equation 3). The filter also appends a default difficulty of "`Dc=0`" to the actual URL so that clients without JavaScript enabled can follow the link while at the same time indicating to the server that they cannot solve work functions.

Before sending the content, the issuing filter updates the Bloom filter to count the request against that client.

4) The Verifying Filter: The verifying filter parses request URLs and extracts any appended proof-of-work variables. If the request URL contains the variables N_c and D_c, they are checked to be current and correct before the module does any computationally expensive operations such as hashing. If N_c and D_c are valid, the verifier then proceeds to check that A satisfies Equation 1. If everything works out, the request is accepted by the high-priority virtual host and the content is sent to the client.

There are three primary reasons why client's request might be rejected by the verifying filter; the URL has no proof-of-work attached, the parameters are not current, or the solution is not valid. The first two failures may have occurred for a variety of legitimate reasons and are not necessarily indicative of a malicious client.

If the request URL contains no PoW parameters, then the client may have been linked to this resource from an external server that has not yet adopted the system and hence did not assign a work function. It is also possible that the user arrived at this website by manually entering the URL into the address bar – users are not expected to know PoW parameters.

If the request URL contains PoW parameters, however they are invalid, the client may have been directed to this site from an external server that appended its own values for N_c and D_c. Alternatively, the user may have taken enough time reading the last webpage that the server has updated it's nonce N_s, invalidating the client nonce N_c as per Equation 3.

Regardless of the reason, once a request is denied, the filter returns an error page to the client, such as the one in Figure 4. The error contains some error text and a single link to the requested content. After it has been processed by the issuing filter, it has a work function embedded into it. The key feature is highlighted; the error page includes an `OnLoad(·)` script that immediately solves the work function and redirects the browser to use the proper URL. The client's web browser history omits the error page, so a user can move through their browsing history without ever seeing this page.

A notable exception is for clients that do not have JavaScript enabled. Recall that the issuing filter embeds "`Dc=0`" into all URLs within HTML tags. If a client's browser does not have JavaScript enabled, it will not solve the work function and instead use the URL verbatim. When the verifying filter observes such a URL with the variable D_c set to zero, it will conclude that the client cannot solve the work function and accept the request on the low-priority virtual host.

D. The Client Solver

While the client end of the system can be computationally demanding, particularly for malicious clients, it is functionally lightweight. The client's browser executes a few scripts contained in the JavaScript file (*kaPoW.js*) linked at the head of the document.

The fundamental script is the `Solve(·)` script which is used to solve individual work functions that the browser encounters. The script takes a tag with a URL as input and extracts the attributes N_c and D_c. Provided it can find those attributes, it systematically hashes them with various values for A until Equation 1 is satisfied. The script removes existing PoW variables embedded in the URL (specifically the "`Dc=0`") and then appends the variables N_c, D_c, and A to the URL. The URL is then updated in the in-memory DOM for use when the browser needs to fetch that resource.

Another script runs as soon as the file is read and hooks into the event triggered when tag elements are added to the DOM. As content tags (such as `<IMG>`) are added, this script calls `Solve(·)` so that the URL in the tag reflects valid work. As hyperlink tags (`<A>`) are added, they have their `ONCLICK` attribute modified to call `Solve(·)` – work functions for hyperlinks are only solved once the user chooses to follow the link.

III. Evaluation

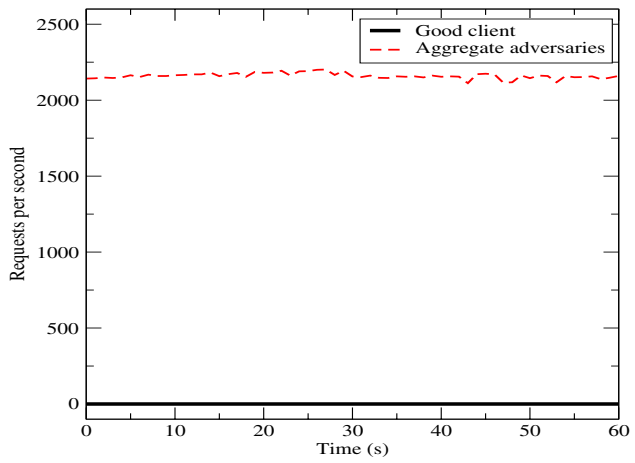
This section evaluates the system using experiments running on a network of 1.8GHz dual processor Intel Xeon machines with Gigabit Ethernet interfaces. The experiments show that the system can efficiently defend against flooding attacks using minimal overhead.

A. Thwarting Flooders

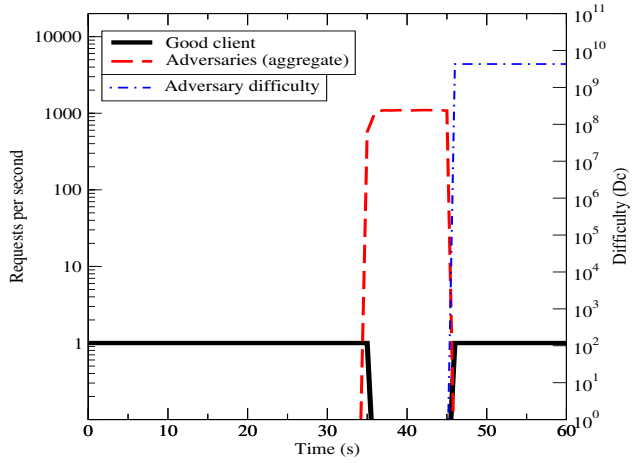
The network has six machines in total; 1 server running Apache, 1 good client which requests a web page once a second and then disconnects, and 4 flooding adversaries that aim to deny the good client by consuming as many server resources as possible. While this setup is far from the magnitude of a real botnet, the server is configured to give adversaries an advantage over the client. Specifically, the server is configured to only accept 4 clients simultaneously and the high-priority virtual host is configured to allow persistent clients to remain connected as long as they continue to send requests.

As shown in Figure 5(a), without `mod_kaPoW` the adversaries can occupy the server indefinitely while they flood packets. The good client can never establish a connection and get a request through.

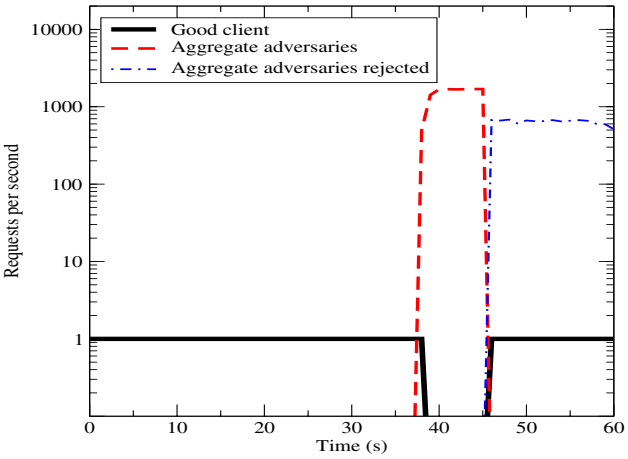
978-1-4244-2219-7/08/$25.00 (c)2008 IEEE



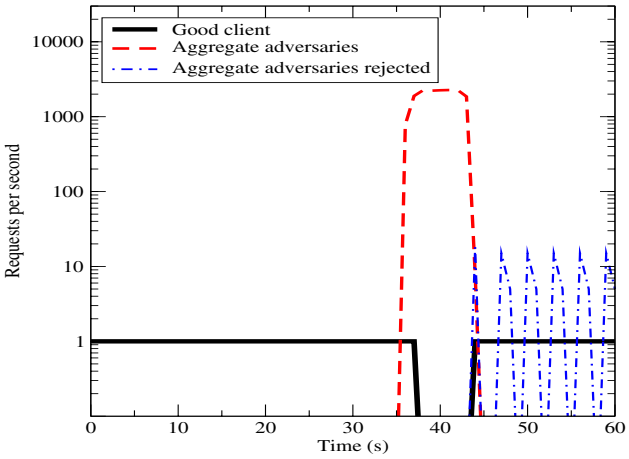
(a) Flooders vs. default server.



(b) Solving Flooders vs. `mod_kaPoW` server.



(c) Flooders vs. `mod_kaPoW` server.



(d) Flooders vs. `mod_kaPoW` server with `iptables` filter.

Fig. 5. `mod_kaPoW` thwarting flooding adversaries.

Figure 5(b) shows the addition of `mod_kaPoW`, limiting the reign of the flooders to a single time window. When the module updates the Bloom Filter, the flooders' requests drive their respective difficulties to 2^{32} (requiring roughly 1.3 hours to solve) preventing them from sending valid requests in time, and restoring access for the good client.

Figure 5(c) shows flooders that do not stop once they are given a difficult work function. Instead, they simply flood requests with invalid solutions. After the initial flood, the adversary requests are rejected and their connections are terminated upon each request, lowering their throughput and allowing access for the good client.

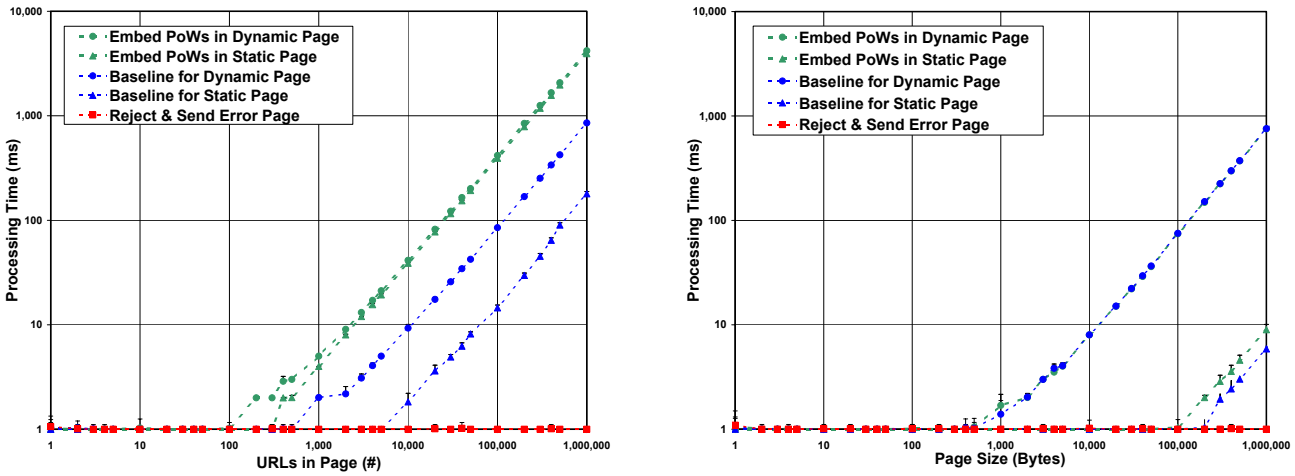
While the system properly discards requests that are part of a flood, they do consume substantial resources on the web server as it must accept each connection, and parse the request before rejecting it. Figure 5(d) shows that using standard `iptables` matching rules, an ingress filter can restrict each client to 5 TCP connections per second. This reduces the magnitude of rejected flooder requests. The drops induce TCP backoff, seen in the jigsaw pattern of rejected requests.

These experiments demonstrate that even using a relatively simple policy to adjust the client difficulty D_c, the system can prioritize clients to achieve separation between determined adversaries and legitimate clients.

B. Overhead

It is important that the overhead imposed on the server by the issuing and verifying filters is minimal. The Apache Benchmark tool (`ab`) was used to measure the time to process a individual web requests, rounding up to the nearest millisecond.

The benchmark in Figure 6(a) shows the overhead when processing files containing a variable number of URLs. Overall, the graph shows that there is observable overhead when processing large static pages stored on the disk. The overhead is considerably less when



(a) Processing time vs. URLs in the file.

(b) Processing time vs. file size.

Fig. 6. Request processing time with respect to the content length.

processing pages dynamically generated because those pages are already chunked appropriately for the filter. When requests are rejected for having an invalid solution, the processing is independent of the file size because the small error page is returned instead. The results indicate that the overhead is negligible for pages containing fewer than a couple hundred URLs, while the benefit of rejecting invalid requests for large pages is substantial.

The benchmark in Figure 6(b) shows the overhead for processing webpages of variable length containing no URLs. The graph shows similar trends to the previous benchmark, although the overhead here is less because there are no calculations of N_c. The benefit of rejecting invalid requests for large pages remains substantial.

IV. Conclusion

This paper presented `mod_kaPoW`, a novel system that is *transparent* to end users like other proof-of-work schemes but is also *backwards compatible* for legacy clients like CAPTCHAs and doesn't require special client software. In the system, a web server dynamically rewrites HTML tags containing URLs by attaching a computational challenge to them. It also sends with the work functions, a small JavaScript so that clients can solve them. Subsequent client requests are prioritized by having a valid solution.

The system binds work functions to the client, server, requested content and time window through the secure creation of the client nonce N_c. The work function difficulty D_c is tailored to each individual client. The evaluation shows that using a simple policy, the difficulty can be adjusted to fend off flooding adversaries. The evaluation also demonstrated the low system overhead.

References

[1] "phpBB: Creating Communites Worldwide," `http://www.phpbb.org`.

[2] R. Stross, "Hannah Montana Tickets on Sale! Oops, They're Gone," *New York Times*, December 2007.

[3] L. von Ahn, M. Blum, N. Hopper, and J. Langford, "CAPTCHA: Using Hard AI Problems for Security," in *Proceedings of Eurocrypt*, 2003, pp. 294–311.

[4] D. Kesmodel, "Codes on Sites 'Captcha' Anger of Web Users," *Wall Street Journal*, May 2006.

[5] S. Hocevar, "PWNtcha," `http://sam.zoy.org/pwntcha`.

[6] "GetAFreelancer.com," `http://www.getafreelancer.com`.

[7] B. Laurie and R. Clayton, "'Proof-of-Work' Proves Not to Work'," in *Workshop on Economics and Information Security*, May 2004.

[8] W. Feng, E. Kaiser, W. Feng, and A. Luu, "The Design and Implementation of Network Puzzles," in *IEEE INFOCOM*, March 2005.

[9] W. Feng and E. Kaiser, "The Case for Public Work," in *Global Internet*, May 2007.

[10] C. Dwork and M. Naor, "Pricing via Processing or Combatting Junk Mail," in *CRYPTO*, August 1992.

[11] T. Aura, P. Nikander, and J. Leiwo, "DoS-Resistant Authentication with Client Puzzles," in *Workshop on Security Protocols*, April 2000.

[12] D. Dean and A. Stubblefield, "Using Client Puzzles to Protect TLS," in *USENIX Security Symposium*, August 2001.

[13] X. Wang and M. Reiter, "Mitigating Bandwidth-Exhaustion Attacks Using Congestion Puzzles," in *ACM CCS*, October 2004.

[14] The Counter, "JavaScript Statistics for October 2007," October 2007, `http://www.thecounter.com/stats/2007/October/javas.php`.

[15] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *ACM Transactions on Networking*, vol. 8, no. 3, June 2000.

[16] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An Improved Construction for Counting Bloom Filters," in *ESA 2006*, September 2006.