mod_kaPoW: Mitigating DoS with Transparent Proof-of-Work

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

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

1. INTRODUCTION

Unwanted traffic remains a fundamental problem for networked systems. Proof-of-work (PoW) is a defense mechanism that adds a client-specific challenge at the start of a networked protocol. 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 set proportional to its relative load on the server.

Recently there have been several proof-of-work systems proposed in the literature [1, 2, 3, 4, 5, 6, 7], although few have actually made much progress towards being deployed. The biggest problems of those schemes is that they require wide-scale adoption of special client and server software in order to operate properly, denying all clients who have not installed the software.

To address this problem, our work investigates a novel web-based proof-of-work system that retains the efficiency of our previous work [1, 2] while focusing on transparency and backwards compatibility for incremental deployment. The system leverages the pervasiveness of JavaScript, software present and enabled on most web-clients, to transparently deliver a challenge, solve it, and submit the client response. The system is designed so that the few clients who do not have JavaScript enabled are not necessarily prevented from accessing the service. The system uses an Apache server module to dynamically embed client-specific challenges in webpages as they are being served. Existing websites do not require changes to any content in order to adopt this system.

This poster presents a prototype of the first proof-of-work system that operates *transparently* and is *backwards compatible* for legacy clients. This shows that proof-of-work systems can be made *incrementally deployable*.

*CoNEXT'*07, December 10-13, 2007, New York, NY, U.S.A.

2. THE PROOF-OF-WORK APPROACH

To a proof-of-work system, all clients are considered adversaries of varying maliciousness that need to be throttled accordingly. In general, a client's maliciousness is measured from the load they have placed upon the system in the past and their contribution to the current load. This value dictates the difficulty of the challenge (called a *work function*) issued to the client; more malicious clients are issued more difficult work functions. It is likely that a well behaving client will be issued a work function of trivial difficulty when the server is not under stress.

The proof-of-work system introduces a new challenge-response step during the initiation of a network protocol. Upon receiving a request, a server *issuer* creates and returns a work function to the client while aborting the connection to avoid storing per-client state. After receiving and solving the function, the client *solver* attaches both the function and solution when resending the request. If the challenge-solution pair is valid, the server *verifier* allows the request to proceed; otherwise the request is denied and another work function may be sent.

The work function has a difficulty tailored to the client. The difficulty is expressed in terms of units of work, where each unit is a uniform computation such as the execution of a hash function. There are many types of work functions with a wide range of properties, and the work function employed determines many of the properties of the system as a whole. All proof-of-work systems must exhibit the following properties to be useful:

- **Efficiency**: Issuing challenges and verifying answers must add minimal overhead, otherwise the system becomes a target for attack.
- **Host Binding**: Work must be bound to the client-server connection to ensure that the specific client is throttled.
- **Time Binding**: The system must resist precomputation and replay attacks for responsive real-time throttling.

Our novel scheme has these properties while introducing the desired properties of transparency and backwards compatibility, unseen in any previous proof-of-work scheme.

3. A TRANSPARENT INSTANTIATION

The prototype protects web content through the addition of work functions to Uniform Resource Locators (URLs) as standard query parameters. While this system could use any of several different types of work functions, the prototype uses the compact Targeted Hash-Reversal function [2]. It is of the form:

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

where H is a one-way uniformly-distributed hash function, N_c is a client-specific nonce generated by the server, D_c is the client-specific difficulty, and A is the solution that the client's solver must find. The parameters are appended to the URL as delimited name-value pairs and their order does not matter. To avoid accidentally triggering an escape sequence, the values are transmitted as hexadecimal values.

The prototype is an Apache module we call `mod_kaPoW` that has two main functions; an *issuing filter* and a *verifying filter*. The issuing filter processes web documents as they are served and embeds work functions in URLs and tags containing URLs, as well as a reference to a JavaScript file with instructions on how to solve the challenges. Figure 1 shows these changes to a simple HTML document.

```
<HTML>
   <HEAD>
      <TITLE>Sample Content Page</TITLE>
      <SCRIPT TYPE='text/javascript' SRC='powurl.js'></SCRIPT>
   </HEAD>
   <BODY>
      <H1>Content!</H1>
      <IMG SRC='test.jpg?Nc=52a6c561&Dc=0' Nc=52a6c561 Dc=10>
   </BODY>
</HTML>
```

Figure 1: HTML file with proof-of-work embedded.

The verifying filter extracts the proof-of-work variables from URLs, and if the parameters N_c and D_c exist, they are checked to be correct. The verifier proceeds to compute the more computationally expensive functions (such as hashing) to check that A satisfies Equation 1. If everything works out, the request is accepted and the desired content is sent.

There are three reasons why a client's request might be rejected by the verifier; the URL has no proof-of-work attached, the parameters are not current, or the solution is not valid. There is one notable exception; when the client uses $D_c = 0$ (i.e. they cannot perform the work) the verifier may accept the request if the server is below capacity. By default all modified URLs contain $D_c = 0$ to accommodate clients without JavaScript. JavaScript-enabled clients will extract the true D_c from the tag and fix the URL before using it.

When a request is denied, the filter returns an error page to the user, shown in Figure 2. The page embeds a browser-reload script along with the challenge so that JavaScript enabled browsers will automatically solve the challenge and follow the link; most clients will not observe this page displayed as it will be reloaded with the content they requested.

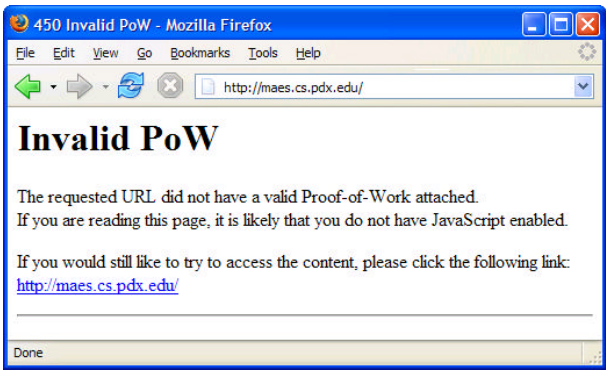

Figure 2: The error page for using an invalid solution.

JavaScript-enabled browsers use the script file provided by the server to solve the embedded work functions. Protected images are solved before a page is loaded so that they render without delay. However, protected hyperlinks are only solved when the user navigates them, to prevent wasting computation on links that will not be traversed.

We argue this is the first proof-of-work system to facilitate incremental deployment because it has two properties not seen in other proof-of-work systems:

- **Transparency:** The system runs without input from users or content developers. Specifically, clients do not need to manually solve proof-of-work and web designers do not need to modify content.
- **Backwards Compatability:** To participate, clients do not need to install software. Those who do not enable JavaScript still retain the ability to access the content, although at lower priority.

4. REFERENCES

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

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

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

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

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

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

[7] B. Waters, A. Juels, J. Halderman, and E. Felten, "New Client Puzzle Outsourcing Techniques for DoS Resistance," in *ACM CCS*, October 2004.