Portland State University

PDXScholar

Computer Science Faculty Publications and Presentations

Computer Science

2007

Protecting the Internet with Public Work

Ed Kaiser
Portland State University

Wu-Chang Feng
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/compsci_fac

Part of the Computer Sciences Commons

Let us know how access to this document benefits you.

Citation Details

Kaiser, Ed and Feng, Wu-Chang, "Protecting the Internet with Public Work" (2007). *Computer Science Faculty Publications and Presentations*. 225.
https://pdxscholar.library.pdx.edu/compsci_fac/225

This Technical Report is brought to you for free and open access. It has been accepted for inclusion in Computer Science Faculty Publications and Presentations by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

Protecting the Internet with Public Work

Ed Kaiser Wu-chang Feng
Portland State University
{edkaiser,wuchang}@cs.pdx.edu

***Abstract*— Distributed denial-of-service attacks represent a growing problem for networked systems. To tackle this problem, this paper explores the addition of a *public* work function to the service advertisement mechanisms used by such systems. When under attack, services advertise this function along with their location information and clients must attach a solution to the function with subsequent requests. The function, which can be made specific to the source of traffic, is publicly verifiable, allowing arbitrary network devices at the edges of the network to quickly verify that subsequent communication from the source will be accepted by the destination. We describe a number of different ways public work can augment current systems and evaluate a promising instantiation of the public work scheme using DNS.**

I. Introduction

Unwanted traffic on the Internet still poses significant challenges. One of the problems is the lack of mechanisms for controlling who or how someone accesses public services. Once its location is known, unsolicited traffic can immediately reach any service. There have been a number of approaches for combating unwanted traffic with solutions ranging from indirection [1], [2], [3], [4], [5], filtering [6], [7], capabilities [8], [9], [10], and proof-of-work [11], [12], [13], [14], [15], [16], [17], [18].

The above systems have salient features that must be combined to adequately address the problem. Indirection provides the ability to hide or dynamically relocate a public service in order to prevent malicious clients from reaching the service indefinitely. Filtering is necessary to stop unwanted traffic as close to the source as possible. Capabilities are necessary to give services dynamic fine-grained control at the request level over who is given access. Finally, proof-of-work is necessary to ensure malicious clients commit as many resources as they are requesting from a service.

In this paper, we describe a system that integrates aspects of each of the techniques into a single mechanism based on the notion of *public* work. The crux of the scheme is simple. The service, when advertising its location, provides a publicly verifiable work function. In order to reach the service, a client must attach a valid answer to this function along with its service request. If the client does not, any network device that has seen the advertisement can verify that subsequent requests are not wanted by the service and can drop them. Since the service itself controls the difficulty of the work function passed back to the clients, it can control access at a fine granularity.

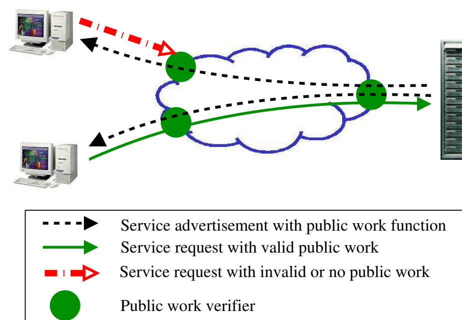

Fig. 1. The public work approach.

There are two significant contributions in this paper. First, we elaborate upon the concept of a *public* work function; a work function with answers that can be verified by machines other than issuer. Second, we describe how a service advertisement mechanism can leverage a public work function to protect its service from unwanted traffic.

II. Basic Approach

Figure 1 shows the basic approach. When advertising its location, a service also supplies a source-specific work function whose solution must be computed by the client and attached to subsequent requests before being given service. The work function can be delivered on-demand (e.g. piggybacked on DNS replies) or a-priori (e.g. via key insertions in a DHT [3], periodic publishing [19], prefetching [20], [21]). The work function is easy to generate and verify, yet adjustably difficult to compute a correct answer. The crucial property is that answers are publicly verifiable, allowing any network device that observes the advertisement to validate subsequent service requests. This enables the edge of the network to detect and filter out traffic that is unwanted by the service. As shown in the figure, requests that lack a valid solution are filtered at the first network device that verifies public work.

The scheme embodies ideas of proof-of-work. The work function given to the client is a puzzle of a certain difficulty. The client must solve the puzzle correctly and attach the answer to subsequent requests in order to reach the service.

The public work scheme draws from indirection by enabling a service to avoid attack by dynamically changing its reachable locations through frequent updates to the public work function. To send a request that will reach the

service, a client must have both the service location and a correct answer to the most recent public work function. Furthermore, the function is source-specific so the service can control the reachable locations on a per-client basis.

The scheme draws on aspects of filtering by supporting destination-controlled filtering at the edges of the network. Specifically, filters can be placed at any point in the network that observes both the advertisement and the subsequent requests. For example, the first-hop router at the client could easily store public work advertisements and check that subsequent requests are valid. A service receiving unwanted traffic would advertise to the sources public work functions with high degree of difficulty or ones without solutions. If a source solves its function, it is slowed considerably. If it ignores the function, intermediate filters will drop its packets.

The scheme draws on capabilities by giving the service complete control over the clients that access it by binding the work function to the requesting client. As before, it can advertise difficult public work functions to clients that it does not wish to hear from and simple ones to clients it does wish to service. Public verification allows the edge of the network to determine if request contains a valid capability.

There are a number of ways that public work could protect Internet services. One could attach public work functions to triggers and keys in I3 and DHTs [2], [3], [22], [23], to beacons in SOS and Mayday [1], [4], to content advertisements in Freenet [24], and to DNS lookups [25]. In this paper, we demonstrate a DNS-based instantiation.

III. Building Blocks

A. Public Work Function

A work function, also known as a cryptographic puzzle, has the signature $\mathcal{W}: \{0,1\}^* \rightarrow \{0,1\}^*$ which relates a puzzle P_D (where D is the puzzle difficulty) to an answer A. To be effective, any machine solving for A given P_D must perform $\Omega(D)$ units of work. In contrast, verifying that A is a solution to P_D must take $O(1)$ units of work.

When any entity can verify that a puzzle-answer pair is valid or not, the work function is considered public. It is possible that many public work functions might exist. We believe the following four properties are necessary for a public work function used within the network:

- Fast issuing: Generating the public work function must add minimal cost to service advertisement.
- Fast verification: While finding a solution to the public work function must be difficult, verifying a solution must add minimal cost to request forwarding.
- Flexible binding: The public work function must be flexible enough to bind to various scales of communication, such as packets, flows, or flow aggregates.
- Limited precomputation and replay: The public work function must prevent precomputation and replay attacks.

To meet the above requirements, our suggested public work function is based on the notion of targeted function reversal. In particular,

$$\mathcal{G}(A, F, N_C, T) \equiv 0 \texttt{ mod } D$$

where F represents flow properties of the subsequent request such as the addresses of the source and destination, T represents the time range on which answers will be accepted, and N_C represents a client specific random number generated by the issuer. The core function $\mathcal{G}$ is any one-way function that has uniformly distributed random output. The function must be one-way so that finding the unknown parameter A involves a non-trivial search. Intuitively, it must also have uniformly random output so that the solver is expected to try D distinct values for A before finding a value that satisfies the equation. In cryptographic terms, $\mathcal{G}$ is assumed to be a random oracle. This work function addresses the requirements above in the following manner:

- Fast issuing: The issuer only needs to generate the random number N_C and a difficulty D to issue a new work function to a particular client.
- Fast verification: A verifier that has recorded N_C and the service advertisement, only needs to compute $\mathcal{G}$ and the modulus to determine the validity of the subsequent communication. Computing the modulus is inexpensive with respect to computing $\mathcal{G}$.
- Flexible binding: The parameters that define F are configurable and include the source and destination addresses, ports, and other protocol fields.
- Limited precomputation and replay: The validity of any particular public work function is directly controlled by the service. Precomputation and replay are limited since the work function is periodically updated through generating and advertising a new random number N_C, as well as the time range T. The time range is included so that the service can expire puzzles more frequently than N_C is updated.

B. Core Function Selection

We investigate the suitability of various cryptographic primitives (block ciphers and hash functions) to be used as the core function in the puzzle. Speed is the primary requirement since puzzle issuers will be faced with a large number of clients. Number theoretic primitives are not considered because they are prohibitively expensive.

Block Cipher vs. Hash Function Puzzle Constructions: In deployment, difficulties will range from 0 to 2^{64} units of work, so D must be an 8-byte integer. To ensure the existence of an answer, A must be at least as large as D. To prevent table-lookup attacks, the random number N_C must also be sufficiently large. The time range T is composed of two 4-byte integers. Finally, the flow identifier F is 12-bytes for the address and port of both the source and destination. In total the input must incorporate 36 bytes.

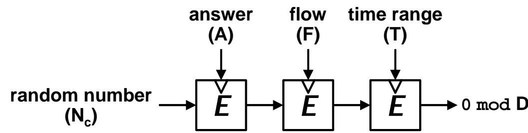

Fig. 2. Puzzle construction using a block cipher. A hatch mark indicates the input that is the key to the cipher.

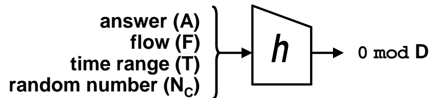

Fig. 3. Puzzle construction using a hash function.

In constructing a block cipher based puzzle, recall that block ciphers have two opposite operations: enciphering and deciphering. If the key were fixed, a one-way construction is not possible since a plaintext can trivially be found that produces the desired ciphertext and vice versa. Consequently, a block cipher based puzzle must use the block cipher's resistance to *key-reversal*. Such a design produces uniform random output under the common assumption that the block cipher E is a secure pseudo-random function.

No block cipher has a large enough key to contain all the puzzle parameters. As a result, the construction sequentially chains cipher calls together. Of the ciphers investigated, the largest key size is 16 bytes, so at least three cipher calls are required to consume the puzzle parameters. The resulting construction is shown in Figure 2. For ciphers with smaller key sizes, more cipher calls are required.

To solve the puzzle, a client selects a candidate answer A, and uses it as a key to encipher the random number N_C. That result is enciphered using the flow identifier F as the key. That result is then enciphered using the time range T as the key. The result is cast as an integer and taken modulo the difficulty D. If the remainder is zero, A is a solution, otherwise the client must restart and try a new value for A.

In contrast to the block cipher based construction, the hash based construction requires the solver to back-out the message that produces a given digest. In particular, this construction exploits the hash function's *preimage-resistance* [26]. The input to each hash function is large enough to consume the concatenation of all the parameters. Each construction uses a single execution of the hash internal compression function, as shown in Figure 3.

Selecting the Construction: Public implementations of the primitives were measured on both Linux (using g++ v. 3.4.2) and Windows (using Visual C++ .NET) 1.8GHz Intel machines. The compiled code was optimized for speed rather than size. The Intel `rdtsc` benchmarking instruction was used to accurately measure the clock cycles to execute each primitive. We indicate on which operating system the fastest measurement for each primitive was obtained.

Table I shows the clock cycles to cipher, decipher, and key various block ciphers. The table is sorted by the cycles incurred to key and run each cipher once, since this corresponds to its use in the construction. For some ciphers, key setup is asymmetric so we report keying for enciphering followed by keying for deciphering. The table shows three suitable ciphers: Tiny Encryption Algorithm (TEA), its strengthened variant XTEA, and Rijndael. Their puzzle constructions must key and call the cipher three times, taking approximately 1800 cycles to compute.

Block Cipher	Key Size (bytes)	Block Size (bytes)	Keying Effort (cycles)	En/Decipher Effort (cycles)	OS
TEA	16	8	–	620/584	Linux
Rijndael	16	16	208/312	408/472	WinXP
XTEA	16	8	–	660/652	Linux
IDEA	8	8	308/2628	472/472	Linux
Twofish	16	16	1304	852/768	Linux
DES	8	8	2400	592/592	WinXP
RC6	16	16	2024	1088/1080	Linux
Serpent	16	16	3420	3872/2780	Linux
3-DES	16	8	3864	3236/3264	Linux
Safer	8	8	5184	3116/2228	Linux
CAST	16	16	8904	1708/1712	Linux
LOKI	16	16	11444	2904/2916	Linux
DEAL	16	16	15268	4152/4180	WinXP

TABLE I
BLOCK CIPHER SPEEDS.

Hash Compression Function	IV Size (bytes)	Input Size (bytes)	Hash Effort (cycles)	OS
MD5	16	64	600	Linux
SHA-1	20	64	1144	WinXP
RIPEMD-128	16	64	1144	Linux
RIPEMD-256	32	64	1380	Linux
RIPEMD-320	40	64	2324	Linux
TIGER	24	64	2496	Linux

TABLE II
HASH COMPRESSION FUNCTION SPEEDS.

Table II shows the clock cycles required to execute the various hash compression functions. The table is sorted by speed and shows MD5 as the leader by a clear margin. The construction invokes the hash compression function once, so the cost of computing the hash based construction is approximately the cost of running the compression function once, which in the case of MD5 is 600 cycles.

Even the slightly slower SHA-1 and RIPEMD-128 constructions are faster than the best block cipher constructions. For speed, we choose a hash based construction over a cipher based construction. However, with a number of public attacks on cryptographic hash functions [27], [28], [29], the reliance of the public work scheme on a hash-based primitive is of concern. This is addressed in several ways.

First, the puzzle design is independent of the hash function used by any particular implementation. As a result, if the hash function selected for the implementation is found to be completely broken (that is, the digest reveals enough information about the message that it is faster to reverse the hash function than solve the puzzle), another hash function can be easily substituted.

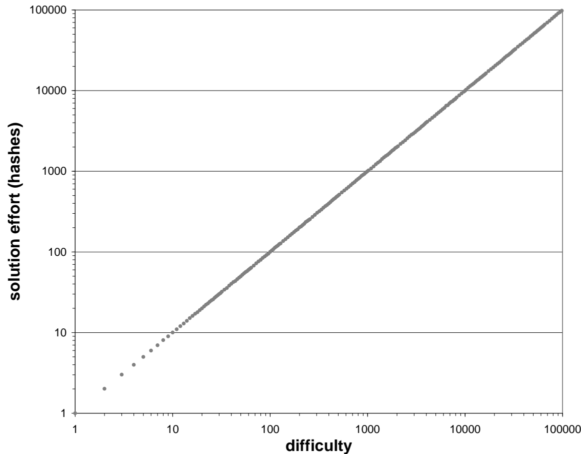

Fig. 4. Average workload for SHA-1 puzzles at various difficulties.

Second, the published attacks target a different property of cryptographic hash functions; the attacks attempt to find two distinct input strings that hash to the same digest (known as a hash-collision). The attacks are only concerned with the digest being the same for both messages. Since the attack is not concerned with finding a message that produces a specific digest (i.e. preimage resistance), the attack has no applicability to the puzzle construction.

Finally, the best attacks still require a large number of hash operations [30]. Of particular note, the best attacks on the collision resistance of MD5 require on the order of 2^{39} MD5 operations, while the best attacks on SHA-1 require on the order of 2^{63} SHA-1 operations. This difference in the collision resistance, if in any way indicative of the hashes' preimage resistance, has led us to choose SHA-1 over MD5.

Uniformly Distributed Output: Although hash functions have not been proven to have uniformly distributed output, experimental evidence indicates that many indeed produce uniformly distributed output [31]. To test our puzzle construction using SHA-1, we randomly generated 10,000 puzzles at each difficulty and then solved them, keeping track of the number of hashes required to find a solution. The average number of hashes to solve a puzzle is linear in the difficulty as shown in Figure 4. This indicates that SHA-1 has uniformly random output and that the construction allows for fine-grained control over the puzzle workload.

IV. DNS-based Instantiation

To showcase a system that uses public work to protect Internet services against unwanted traffic, we designed and built an instantiation using modifications to the current DNS lookup and advertisement mechanisms.

A. Assumptions

It is important to understand some underlying assumptions that drive aspects of our instantiation. In particular, our approach assumes the following:

- *Source spoofing*: We assume that spoofing source addresses is still possible [7], [32], but that flooding attacks using spoofed source addresses are rare. Specifically, the proliferation of ingress filtering [33], [34], [35] has reduced opportunities for spoofing. In addition, since compromised machines have significant economic value to their owners [36], [37], [38], they typically do not attempt activity that would attract attention such as sourcing a packet flood with spoofed addresses.
- *Blind spoofing of DNS advertisements difficult*: We assume that it is difficult for a machine other than the authoritative name server to produce a valid DNS reply. That is, the underlying advertisement mechanism is secure and that appropriate measures have been taken to eliminate DNS cache poisoning and blind spoofing attacks [39], [40].
- *Path between client and server is not completely asymmetric*: We assume that there exists an enforcement point at the client edge (such as the first-hop router, a DSL/cable modem or an access link) that is on the path both to and from the client and server. This is to ensure that the verifier has the ability to store public work advertisements and to validate subsequent replies. Note that, asymmetry such as multihoming can be accommodated if secure sharing of collected public work advertisements is supported.

B. DNS Lookup

Figure 5 shows the general approach which modifies the DNS lookup mechanism commonly used. Typical DNS lookups start with a client issuing a recursive query to their local DNS cache for the name being resolved. If the cache does not already have the record, it iteratively queries the DNS hierarchy until it reaches the authoritative name server which returns the result.

In our instantiation, rather than issue a recursive query to the local DNS cache for the name, the client issues a recursive query for the *authoritative server* of the name (i.e. the NS record). Upon receiving the location of the authoritative server, it then issues an iterative query to the authoritative server to get the location and the public work function associated with it. This allows the issuer to send the client-specific public work function directly to the client, preventing malicious agents that are not along the path from obtaining the work function for a client and then spoofing requests from that client. Since the reply containing the public work function is sent directly from the issuer to the client, any network device along this path can note the work function and verify subsequent requests satisfy the function. As this requires both path symmetry and per-flow state to be kept to work, this should be done as close to the client as possible such as at the first-hop router.

The approach retains a significant amount of the caching benefit within the DNS hierarchy, namely all of the inter-

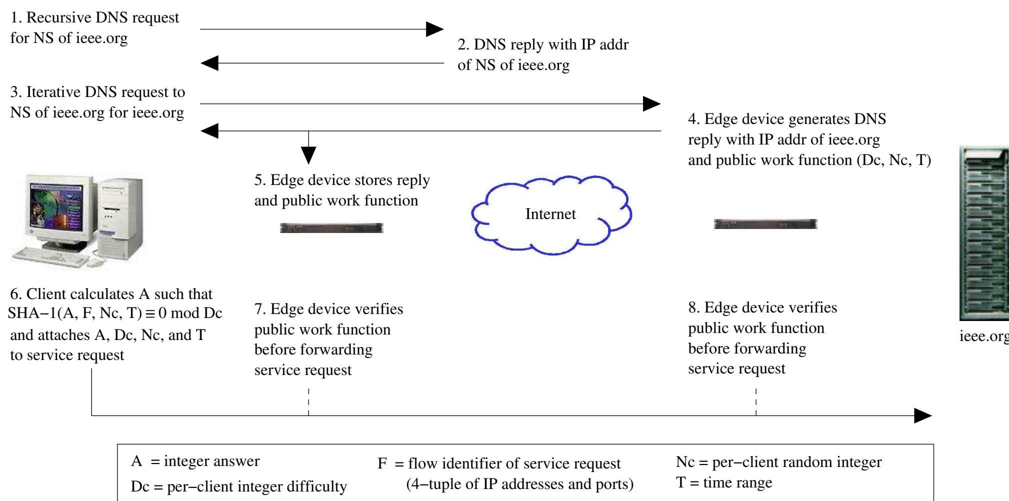

Fig. 5. A DNS-based instantiation of public work.

mediate lookups, however, the final resolved name cannot be cached [41]. While this can lead to more load on the authoritative servers, there are several mitigating factors. The first is that the DNS lookup operation on the authoritative server has very low overhead and is easily scaled. The name database itself can be placed entirely in memory and the process is easily parallelizable [42]. The second is the increasing use of CDNs to distribute names has reduced the latency difference between serving a name out of the local DNS server cache and serving a name from the authoritative server [43]. In addition, many of the low-level DNS entries themselves have short TTLs to support responsive failover for production web services. As shown in several DNS studies [44], [45], [46], a policy of reducing the TTL of advertisements drives down DNS cache hit rates substantially. Even with such mitigating factors, a measurement study will be required to understand the scheme's impact on DNS caching performance given the penchant for local DNS servers to ignore DNS TTLs [47].

C. Work Management

From an economic standpoint, in order for a proof-of-work scheme to be effective, the amount of work done by the "good guys" and the amount of work achievable by the "bad guys" must differ significantly [48]. In particular, due to the sheer number of compromised systems that exist today, a global difficulty setting is easily overcome by brute force. Stated slightly differently, no proof-of-work system can function properly unless difficulties are properly tailored based on the usage history of client.

There are ways to tailor work function difficulty to clients by using information from reputation systems, black lists, packet queue heuristics, and resource consumption records. Unfortunately, such an intricate control mechanism is outside the scope of this paper. In our implementation, client difficulties are proportional to the requests they send to the service.

D. Attack Vectors

It is helpful to analyze the protection provided by the DNS instantiation of public work by looking at a number of attack vectors that a set of malicious machines might employ against it.

Request flooding with valid public work: With the sheer amount of resources available in large Botnets, the attacker could attempt to solve the public work given to each of the zombie clients. Given a proper strategy for setting the difficulty based on usage, the public work functions assigned to the zombies will become harder, eventually to a point where zombie clients cannot flood packets fast enough to overwhelm the service. In addition, the work function essentially pinpoints the zombies sourcing the traffic, drawing attention to the compromised machines.

Request flooding without valid public work: The attacker could ignore all public work advertisements it receives and attempt to send service requests without performing work. In this case, any verifier at the client edge that has recorded the public work function, can efficiently determine that the request traffic would eventually be filtered at the destination and drop it early. At worst, if no device near the client enforces public work, a device at the server's edge will perform the verification and drop the requests at the entry point to its network.

Flooding issuer with DNS requests: The attacker can attempt to disable the service by disabling the issuer via flooding it with DNS requests. This is difficult due to the efficiency of the lookup mechanism itself. In particular, the the lookup is easily parallelizable, the replies consume a modest amount of bandwidth, and the name database can easily be stored in memory [42]. Flooding attacks using spoofed DNS requests will then force the attacker to generate large amounts of spoofed traffic which are counter to our first assumption. If such attacks are possible, however, one could mitigate its effect by applying public work to the DNS lookups going to the authoritative name

server by attaching a work function to the NS record.

Spoofing the issuer: The attacker can attempt to forge the service advertisement and the public work function itself via a DNS poisoning attack [39]. In this case, our system relies on the security of the current DNS lookup to ensure this is difficult. Proper filtering of replies and the use of DNSsec [40] eliminate this avenue of attack. In fact, much larger problems exist if the adversary lies along the path between clients and the service, or if traffic can be redirected away from the legitimate service.

Spoofing work: The attacker can target a single client by using the zombie network to perform large amounts of public work using the victim's public work function. The attacker can then spoof subsequent requests in order to inflate the resource consumption and difficulty for the victim. This attack is rendered useless since the public work advertisement is sent directly from the issuer to the client. The attacker would have to lie somewhere along the path between the client and server to to capture the client-specific public work function.

Spoofing DNS requests: The attacker can target a single client by repeatedly spoofing DNS requests from it to the authoritative name server. While the victim will receive the valid service advertisement and public work function, it will be forced to drop it since it did not request the information and can not tell if it is valid. This attack vector can be used for multiple purposes. The attacker could disable all services for the victim by spoofing DNS requests to all of the services the victim wishes to contact in advance or the attacker could disable a particular service by spoofing DNS requests from all clients going to that service. Both require large amounts of spoofed traffic to be sent which is counter to our assumptions. While there are several ways to mitigate this attack, perhaps the cleanest is to rely on DNSsec. Rather than drop the service advertisement and public work function generated by a spoofed DNS request, the victim can verify that the advertisement is valid and keep it for future use. Other mechanisms that mitigate this problem include adding public work to the lookup at the authoritative name server or rate limiting DNS requests (especially duplicate ones) per host at the network ingress.

V. Implementation

The implementation involved changes to the open-source name server BIND [49]. In total, only 39 lines of code were modified to issue a new record type. The public work verifier was implemented as an `iptables` module that hooks into the Linux network chains. Answers are packed into the sequence and acknowledgement numbers of the SYN packet. To facilitate this, a client puzzle solver was also implemented as a network module. Both modules were implemented in under 200 lines of code. [Note to reviewers: All code associated with the implementation will be made publicly available]. The implementation details are described below:

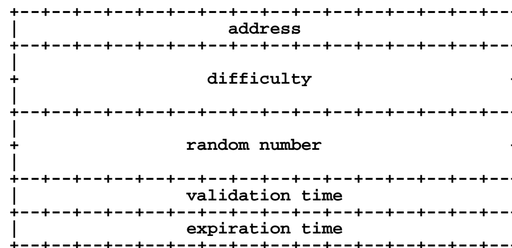

Fig. 6. Format of the DNS puzzle record. The width is 32 bits.

A. Authoritative DNS Server

To support a public work function in the BIND name server `named`, a new puzzle (PUZ) record type was added. This record returns the service address similar to an A record but also includes the client-specific parameters of the work function: the difficulty D_C, the random number N_C, and the time range T (expressed as a start time and expiration time). The structure of the RDATA portion of the record is shown in Figure 6.

This record type involved modification of the BIND database loading routine which is only executed at start up to mark service names that it require a work function to be advertised with the location. The database searching routine was modified to return a stub puzzle record (only containing the address) when it finds a PUZ entry for the service. The response generating routine was modified to append the parameters D_C, N_C and T to the stub puzzle record. This routine is where D_C and N_C are looked up. No further modifications are necessary when the structure is passed to the functions for formatting, appending authority information, and finally sending the datagram.

B. Verifying Router

This `iptables` module is designed to be placed at either edge of the network and observe PUZ records as they pass by. Later, it verifies the subsequent SYN packet of flows corresponding to the PUZ records. The router uses a hash table with 100,000 entries to store the parameters and IP portions of the flow identifier. Recall that the PUZ record is sent from the DNS authority directly to the client; so the client address is the destination address while the service address is extracted from the PUZ record.

When the router observers the subsequent SYN packet sent from the client to the service, it fetches the stored D_C, N_C, and T. If the parameters are fresh (i.e. the time is within T), the router takes the answer (stored in the sequence and acknowledgement numbers) and SHA-1 hashes it with parameters, and checks that the digest is congruent with zero modulo D_C. If it is, the answer is correct and the packet is forwarded. Otherwise if the answer is wrong, the packet is dropped.

C. Client Puzzle Solver

Similar to the verifying module, the client puzzle solver observes PUZ records sent to clients in its network, recording the parameters D_C, N_C, and T of the work function

Sub-Process	Unmodified BIND (cycles)	Modified BIND (cycles)
Receive Request	8130 ± 2368	9254 ± 2503
Parse Request	1500 ± 808	1684 ± 850
Search Database	13702 ± 3459	13720 ± 3654
Create Reply	4976 ± 1662	4561 ± 1717
Add Auth Info	31627 ± 5510	32906 ± 5632
Send Reply	110079 ± 9450	109058 ± 9411
Total	170013 ± 17498	171183 ± 16512

TABLE III

TIMING ANALYSIS OF DNS REQUEST PROCESSING.

with the appropriate flow. These parameters are stored in a hash table similar to the one used in the verifying router.

When the solver observes a TCP SYN packet corresponding to a previously observed PUZ record, the solver begins the search for a solution. As stated earlier, the resulting answer is packed into the sequence and acknowledgement numbers in the SYN packet. While this avoids adding an IP option (which would cause some routers to process the packet in the slow path) and appears transparent to the end hosts, it adds complication in that the sequence numbers of all following packets must be translated as to not disrupt the TCP state at either end. This masquerading requires a second table to store the precise flow (addresses and ports of both the client and server) as well as the delta between the new sequence numbers and the original. To minimize the delta, the solver begins the search for the answer from the original sequence numbers.

Finally, the client DNS resolver had to be modified to directly query the authoritative DNS server and accept the PUZ response. These changes were very minimal and added 10 lines of code.

D. Benchmarking the system

We measured each component of the system individually. The measurements were taken on Linux 1.8GHz Intel machines using g++ v. 3.4.2 and the `rdtsc` benchmarking instruction. The machines have gigabit Ethernet cards.

The first component measured was the authoritative name server. Table III summarizes the cycles required to compute each subroutine for 1 million requests before and after modification. As the table shows, the modified routines are minimally impacted. Specifically, the overhead within those routines are within the standard deviation of each other. Further, the database and searching routines, which contain most of the modifications, comprise a small fraction (approximately 11%) of the total time required to process a DNS request. Not surprisingly, the most expensive subroutine (one not directly altered) is the one for putting the datagram on the wire. Even though we are sending a slightly larger response, the time to execute this bottleneck is not noticeably impacted.

The total time required to process a request is around $100\mu s$ on the 1.8GHz machine, meaning that a single authoritative DNS server can process 10,000 DNS requests per second. Since DNS is very parallelizable, load-balancing amongst several DNS authorities can increase lookup throughput and thus work function issuance. Load-balancing, if done by IP range, has the additional benefit of reducing the range of addresses that each issuer will see, as a result one could expect to see speed improvements in establishing per-client difficulty.

Sub-Process	Effort (cycles)
Record Work Function	913 ± 257
Verify Answer	2744 ± 1976

TABLE IV

TIMING ANALYSIS FOR THE NETWORK VERIFIER.

Sub-Process	Effort (cycles)
Record Work Function	2055 ± 2012
Store Answer	2614 ± 2447
Translate Sequence Numbers	1013 ± 690

TABLE V

TIMING ANALYSIS FOR THE CLIENT SOLVER.

Next we measured the verification router module. Table IV summarizes the cycles to record puzzles and verify TCP SYN packets. The table shows that the function for recording puzzles is extremely fast. We believe this is due to this module only having one table to access, resulting in good cache performance. The function for verifying answers is somewhat more expensive since it must compute one hash. These times correspond to $0.5\mu s$ to record a work function and $1.5\mu s$ to verify a SYN packet. We believe the verification operation is fast enough (650,000 verifications per second) for a client edge device since it has limited hosts it is working for.

Finally we measured the client solver module. Table V summarizes the cycles to record puzzles, translate sequence numbers (in either direction), and the base cost of recording a puzzle answer. We notice that the effort to record puzzles and store answers is more than that required of the verifier. We believe this is due to cache misses since we are manipulating twice as much memory here. Note that the timing of this function omits the time it takes the client to solve the puzzle because it varies and is intended to be large for bad clients. We are pleased to observe that the translation function is inexpensive (just over $0.5\mu s$) since it is used most frequently.

VI. RELATED WORK

SOS [1], Mayday [4]: SOS and Mayday use indirection to dynamically hide the location of the service being protected. Note that, in a sense, layered names [50], scheduled names [51], and HIP [52] provide similar functionality, but their use in mitigating targeted attacks has not been as clearly defined. The public work system differs from these schemes in its use of proof-of-work as part of the indirection. In addition, rather than relying on a centralized set of beacons to perform the validation and filtering, the public work scheme allows multiple intermediate nodes along the

path to perform the functions. In its DNS instantiation, the public work system also does not rely on an overlay network. Public work functions are a complementary feature that could be added to the overlay schemes, however. For example, a public work function could be used as a Mayday authenticator, which when propagated throughout the ring, allows traffic shedding at the edges.

i3 [3]: The i3-based scheme uses the notion of public and private triggers for accessing services. The public trigger is used to negotiate the parameters of the private trigger. i3 adds a proof-of-work mechanism to protect the private trigger mechanism. With the public work scheme, the service advertisement is equivalent to the public trigger and the subsequent answer to the public work function serves as a private trigger. The two key differences are that the public work issuer can validate requests using a constant amount of state rather than having to rely on a large puzzle database and that verifiers can be placed anywhere along the path that sees both the service advertisement and the subsequent requests. A straight-forward extension to i3 would be to propagate a public work function back through the overlay. The overlay node that is directly connected to the source would then store the public work function in order to validate subsequent communication.

Network Address Space Randomization (NASR) [5]: Like NASR, the public work approach is inspired by the application of biological diversity into computer systems [53], [54], [55], [56]. Rather than directly modifying the actual location of the service, however, the public work scheme randomizes its "effective" location. In a sense, the true location of a service can be seen as the dynamic concatenation of an answer to a recent public work function along with the destination address and port.

Pushback [6]: The public work scheme is similar to Pushback in that it allows for upstream filtering all the way back to the source of attack. Whereas Pushback supports binary filtering, the public work scheme provides an analog filter which can be tuned by changing the difficulty setting of the work function. One important difference is that source-based Pushback filters are amenable for use throughout the network while the public work verifiers are suitable only at the edges. In particular, at the issuer, the verifier runs with a constant amount of state while at any other point in the network, the verifier is required to keep per-flow state in the form of service advertisements.

DNS blacklists [57]: DNS-bl is a cooperative effort amongst DNS providers to deny DNS service to known spam domains. Its power is in its ability to take distributed reports of abuse and to aggregate them into a database that any DNS server can use to keep spammers from correctly resolving names. The public work scheme takes a similar approach, but integrates proof-of-work to obtain a dynamic, analog control over malicious sources. The public work approach could benefit significantly by using the aggregated reports provided by DNS-bl.

Capabilities [8], [9], [10]: Capability systems provide the end service fine-grained control over who is given access. In a sense, the service advertisement containing the public work function is a capability. Public work can augment capabilities by adding proof-of-work so that malicious clients cannot target multiple services simultaneously and by allowing capabilities to be publicly verified rather than relying on the issuer for this. Providing early enforcement of capabilities stops malicious activity closer to the source.

Proof-of-work [11], [12], [13], [14], [15], [16], [17], [18]: Public work is an instantiation of proof-of-work and client puzzles. Unlike previous approaches, the public work scheme works across all named services and not just for a single protocol or application. One exception is network layer puzzles. The public work scheme differs from network layer puzzles in its use of public verification. In particular, the puzzle issuer and puzzle verifier no longer have to be co-located. Perhaps closest to the public work scheme is Hashcash [14], a publicly verifiable proof-of-work system that targets spam. In Hashcash, global random numbers (the lottery numbers of the previous day) are used to produce work that can be verified by any mail server. Our approach differs in that it is service-based: the service itself generates and broadcasts the public work function. In addition, Hashcash only protects against spam.

VII. Conclusions and Future Work

While our instantiation indicates that public work is a promising approach for combating the problem of unwanted traffic, there are many issues that still need to be addressed. As described earlier, one area that needs to be more fully examined is the impact that the specific DNS modifications have on network performance. In particular, since per-client public work functions effectively disable DNS caching for the final name lookup, it is important to measure the impact such a change will have on lookups to the authoritative servers. Another area that must be addressed is the performance of the system while it is under attack by sophisticated adversaries. Our end goal is to develop an algorithm for managing work functions so that the optimal strategy for the adversary is to act like an average client. While developing these local algorithms and policies is important, another interesting avenue for future work is to use federated reputation systems such as DShield or DNS-bl to drive work functions [57]. Using such coordination prevents an adversary from targeting multiple sites. Finally, while targeted function reversal has many salient features, it is only one of many possible public work functions that could be used. We plan on examining and comparing alternative public work functions for use in our system.

References

[1] A. Keromytis, V. Misra, and D. Rubenstein, "SOS: Secure Overlay Services," in *ACM SIGCOMM*, August 2002.

[2] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet Indirection Infrastructure," in *ACM SIGCOMM*, August 2002.

[3] K. Lakshminarayanan, D. Adkins, A. Perrig, and I. Stoica, "Taming IP Packet Flooding Attacks," in *HotNets*, November 2003.

[4] D. Andersen, "Mayday: Distributed Filtering for Internet Services," in *USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[5] S. Antonatos, P. Akritidis, E. Markatos, and K. Anagnostakis, "Defending Against Hitlist Worms Using Network Address Space Randomization," in *Workshop on Rapid Malcode*, November 2005.

[6] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling High Bandwidth Aggregates in the Network," *ACM SIGCOMM CCR*, vol. 32, no. 3, July 2002.

[7] M. Handley and A. Greenhalgh, "Steps Toward a DoS-resistant Internet Architecture," in *ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, August 2004.

[8] T. Anderson, T. Roscoe, and D. Wetherall, "Preventing Internet Denial-of-Service with Capabilities," in *HotNets*, November 2003.

[9] Z. Duan, K. Gopalan, and Y. Dong, "Push vs. Pull: Implications of Protocol Design on Controlling Unwanted Traffic," in *SRUTI*, July 2005.

[10] X. Yang, D. Wetherall, and T. Anderson, "A DoS-limiting Network Architecture," in *ACM SIGCOMM*, August 2005.

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

[12] A. Juels and J. Brainard, "Client Puzzles: A Cryptographic Defense Against Connection Depletion," in *NDSS*, February 1999.

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

[14] A. Back, "Hashcash: A Denial of Service Counter-Measure," Tech. Rep., Cypherspace, August 2002, `http://www.hashcash.org/papers/hashcash.pdf`.

[15] X. Wang and M. Reiter, "Defending Against Denial-of-Service Attacks with Puzzle Auctions," in *IEEE Symposium on Security and Privacy (S&P)*, May 2003.

[16] W. Feng, "The Case for TCP/IP Puzzles," in *ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, August 2003.

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

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

[19] M. Handley and A. Greenhalgh, "The Case for Pushing DNS," in *HotNets*, 2005.

[20] E. Cohen and H. Kaplan, "Prefetching the Means for Document Transfer: A New Approach for Reducing Web Latency," in *IEEE INFOCOM*, 2000.

[21] V. Ramasubramanian and E. Sirer, "The Design and Implementation of a Next Generation Name Service for the Internet," in *ACM SIGCOMM*, August 2004.

[22] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *ACM SIGCOMM*, August 2001.

[23] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," in *ACM SIGCOMM*, August 2001.

[24] I. Clarke, O. Sandberg, B. Wiley, and T. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," *Lecture Notes in Computer Science*, vol. 2009, 2001.

[25] P. Mockapetris, "Domain Names: Concepts and Facilities," November 1983, RFC 882.

[26] P. Rogaway and T. Shrimpton, "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance," in *Fast Software Encryption*, February 2004.

[27] E. Biham and R. Chen, "Near-Collisions of SHA-0," in *CRYPTO*, August 2004.

[28] X. Wang and H. Yu, "How to Break MD5 and Other Hash Functions," in *EUROCRYPT*, May 2005.

[29] V. Klima, "Finding MD5 Collisions: a Toy for a Notebook," Tech. Rep., March 2005.

[30] I. Mironov, "Hash Functions: Theory, Attacks, and Applications," Tech. Rep., Microsoft Research, October 2005.

[31] M. Bellare and T. Kohno, "Hash Function Balance and its Impact on Birthday Attacks," in *EUROCRYPT*, May 2004.

[32] R. Morris, "A Weakness in the 4.2BSD Unix TCP/IP Software," Tech. Rep., AT&T Bell Laboratories, February 1985, Bell Labs Computer Science Technical Report 117.

[33] P. Ferguson and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing," *RFC 2827*, May 2000.

[34] F. Baker and P. Savola, "Ingress Filtering for Multihomed Networks," March 2004.

[35] R. Beverly and S. Bauer, "The Spoofer Project: Inferring the Extent of Internet Source Address Filtering on the Internet," in *SRUTI*, 2005.

[36] S. Staniford, V. Paxson, and N. Weaver, "How to 0wn the Internet in Your Spare Time," in *USENIX Security Symposium*, August 2002.

[37] S. Kandula, D. Katabi, M. Jacob, and A. Berger, "Botz-4-Sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds," in *NSDI*, May 2005.

[38] D. Kawamoto, "Blackmailers Try to Black Out Million Dollar Homepage," January 2006, `http://news.com.com/2100-7349_3-6028131.html`.

[39] K. Huagsness, "DNS Cache Poisoning Detailed Analysis Report, Version 2," March 2005, `http://isc.sans.org/presentations/dnspoisoning.php`.

[40] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, "DNS Security Introduction and Requirements," March 2005.

[41] C. Wills and H. Shang, "The Contribution of DNS Lookup Costs to Web Object Retrieval," Tech. Rep. WPI-CS-TR-00-12, 2000.

[42] T. Deegan, J. Crowcroft, and A. Warfield, "The Main Name System: An Exercise in Centralized Computing," *ACM SIGCOMM CCR*, vol. 35, no. 5, October 2005.

[43] Akamai Technologies, "Fast Internet Content Delivery with FreeFlow," Tech. Rep., Akamai, Inc., April 2000.

[44] H. Balakrishnan J. Jung, E. Sit and R. Morris, "DNS Performance and the Effectiveness of Caching," in *ACM SIGCOMM Internet Measurement Workshop*, November 2001.

[45] J. Jung, A. Berger, and H. Balakrishnan, "Modeling TTL-Based Internet Caches," in *IEEE INFOCOM*, 2003.

[46] E. Cohen and H. Kaplan, "Performance Aspects of Distributed Caches Using TTL-based Consistency," in *ICALP*, 2001.

[47] J. Pang, A. Akella, A. Shaikh, B. Krishnamurthy, and S. Seshan, "On the Responsiveness of DNS-based Network Control," in *Internet Measurement Conference*, October 2004.

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

[49] Internet Systems Consortium, "Bind 9.3," `http://www.isc.org/index.pl?/sw/bind/bind9.php`.

[50] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish, "A Layered Naming Architecture for the Internet," in *ACM SIGCOMM*, August 2004.

[51] J. Crowcroft, "Scheduled Name Space," `http://www.postel.org/pipermail/end2end-interest/2004-April/003921.html`, April 2004.

[52] IETF Working Group, "Host Identity Protocol (HIP) Charter," `http://www.ietf.org/html.charters/hip-charter.html`.

[53] S. Forrest, S. Hofmeyr, and A. Somayaji, "Building Diverse Computer Systems," in *HotOS*, May 1997.

[54] P. Szor and P. Ferrie, "Hunting for Metamorphic," in *Virus Bulletin Conference*, September 2001.

[55] PaX Team, "PaX Address Space Layout Randomization (ASLR)," `http://pax.grsecurity.net/docs/aslr.txt`.

[56] T. Durden, "Bypassing PaX ASLR Protection," *Phrack*, , no. 59, July 2002.

[57] "DNS Providers Blacklist," `http://www.dnsbl.org`.