New Client Puzzle Outsourcing Techniques for DoS Resistance

Brent Waters[1], Ari Juels[2], J. Alex Halderman[1], and Edward W. Felten[1]

[1] Princeton University
Princeton, NJ
{bwaters,jhalderm,felten}@cs.princeton.edu

[2] RSA Laboratories
Bedford, MA
ajuels@rsasecurity.com

ABSTRACT

We explore new techniques for the use of cryptographic puzzles as a countermeasure to Denial-of-Service (DoS) attacks.

We propose simple new techniques that permit the *outsourcing* of puzzles—their distribution via a robust external service that we call a *bastion*. Many servers can rely on puzzles distributed by a single bastion. We show how a bastion, somewhat surprisingly, need not know which servers rely on its services. Indeed, in one of our constructions, a bastion may consist merely of a publicly accessible random data source, rather than a special purpose server. Our outsourcing techniques help eliminate puzzle distribution as a point of compromise.

Our design has three main advantages over prior approaches. First, it is more resistant to DoS attacks aimed at the puzzle mechanism itself, withstanding over 80% more attack traffic than previous methods in our experiments. Second, our scheme is cheap enough to apply at the IP level, though it also works at higher levels of the protocol stack. Third, our method allows clients to solve puzzles offline, reducing the need for users to wait while their computers solve puzzles.

We present a prototype implementation of our approach, and we describe experiments that validate our performance claims.

Categories and Subject Descriptors

E.3 [**Data**]: [Data Encryption]

General Terms

Security

Keywords

Denial-of-Service, DoS, Client Puzzles

CCS'04, October 25-29, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-961-6/04/0010 ...$5.00.

1. INTRODUCTION

Denial-of-service (DoS) attacks present a strong and well established threat to the Internet and e-commerce. One proposed countermeasure requires clients to commit resources to an interaction by successfully solving a computational problem known as a *client puzzle* [16, 23] before a server will provide resources to the client. This prevents an attacker from consuming a large portion of the resources of a targeted server without commanding and investing considerable resources himself.

1.1 Shortcomings of Existing Solutions

While the deployment of client puzzles in attack scenarios seems promising, we have found that most proposed systems of this type have two basic shortcomings.

The first is that the client puzzle mechanism itself can become the target of a denial-of-service attack. In most systems either the puzzle creation or verification operation (or both) require the server to perform a cryptographic hash computation [23, 6, 13]. This opens the possibility that the puzzle verification mechanism itself will be the target of a denial of service attack, in which an attacker floods the server with bogus puzzle solutions that the server has to process. Thus existing client puzzle mechanisms replace one possible DoS attack with another. Although the DoS attack on the puzzle mechanism requires more attack resources than before, this is still not an ideal situation. The experiment we present in Section 4.1 demonstrate that puzzle verification increases the server's processing time per new TCP connection request by approximately 80 percent with existing solutions.

A few systems [2] attempt to alleviate this problem by outsourcing the hash computation to a designated gateway, but this merely pushes the vulnerability to a different target. Furthermore, a gateway in these systems needs to be aware of each server it might service and thus will be difficult to scale. Deploying a robust gateway service in this manner seems infeasible.

The second shortcoming in current solutions is that clients must, in practice, solve them in an on-line fashion. For example, if a website employs client puzzles, then a user who wants to visit the site has to wait for his computer to solve a puzzle before accessing the site. Thus puzzles use up not only computer time, but also users' time, which is often much more valuable. Since many users have little patience for website delays, a site that imposes long puzzle delays can drive away legitimate users.

This puts the adversary at a cost advantage. He is not concerned with whether there are human operators at the machines he employs for his attack. This means that a puzzle that costs the attacker some fixed price to solve will cost legitimate clients much more, due to the higher cost of human time for real clients. (Some sites require human intervention, by using CAPTCHAs [35], but that raises other issues.)

1.2 Our Solution

In this paper, we present a new way to use puzzles to mitigate denial-of-service attacks. Our solution has three main attributes:

- The creation of puzzles is outsourced to a secure entity we call a bastion. An arbitrary number of servers can use the same bastion, and can safely share the same set of puzzles, due to special cryptographic properties of the puzzles. Once constructed, the puzzles will be digitally signed by the bastion so that they can be redistributed by anyone.

- Verifying a puzzle solution requires very little work for a server. In fact, it only requires a simple table lookup.

- Clients can solve puzzles off-line, so that users do not have to wait for puzzles to be solved.

- Solving a puzzle gives a client access, for a time interval, to a "virtual channel" on the server—i.e., to a small slice of the server's resources—and the server ensures no virtual channel uses more than its fair share of available resources.

Previous schemes involve puzzle distribution on a per-request or per-session basis. Our approach is more coarse-grained in that it relies on virtual channels, which can be used as an abstraction to protect different types of resources. For example, a web server might limit the number of open TCP connections per channel or a database server could control the rate of database queries processed. When at high risk of DoS attack (or in the midst of an attack) a host in our system accepts communication only via a restricted collection of channels.

To contact a host through one of these channels, a client must provide a valid token. A token consists of the solution to the client puzzle associated with a particular channel and time interval. A client can easily attach tokens to every packet it transmits. The host can enumerate in advance the set of valid tokens, so the host can verify tokens and filter channel traffic very efficiently. An adversary with limited computational resources can successfully attack only a limited number of channels, and the remaining channels will be available to support normal communications from benign clients. We note that multiple clients can use the same channel for communication. The primary purpose of channels is to segregate adversary requests from user requests.

We justify the use of tokens by the following observation. In typical DoS attacks an attacker commandeers a cohort of "zombie" machines on the edge of the network, but generally does not compromise routers in the middle of the network. Based on this observation, we consider an attack model that assumes only limited eavesdropping by the adversary. (This assumption is explored further in Section 5.3.)

As we have explained, puzzle-based DoS solutions provide a newly attractive DoS target: the point of distribution of puzzles. To address this problem, we propose a novel approach to client-puzzle distribution. We show how to *outsource* puzzle distribution to a robust independent web service such as a highly distributed content-serving network like Akamai or a well-protected set of core servers like the root DNS system. We refer to this service as a *bastion*. A bastion will serve as a leverage point, reducing the robustness requirements needed to defend a server against DoS.

We present three methods for outsourcing puzzle distribution, each with different requirements for the bastion and defending servers. Our preferred "D-H" construction, which is based on the Diffie-Hellman problem, has two important properties that allow it to avoid the shortcomings of previous client puzzle systems. The first is that a server's puzzle solutions are made from a combination of the server's public key and the solution to a puzzle posed by the bastion. When publishing a puzzle, the bastion does not need to know which servers will use that puzzle. Since servers can effectively share puzzle challenges, only a constant number of puzzles need to be published for each time interval, and these puzzles can be distributed and replicated widely. This property, along with the ability to quickly check token solutions, insulates the puzzle distribution mechanism from attack.

The second property is that when a client solves a puzzle for a particular channel, the solution can be used at any server. The solution for a particular channel is combined with the public key of a server to produce a token solution specialized for that server. This means the client machine can compute solutions ahead of time and adapt them on the fly to whatever servers the user chooses to contact. The user will then experience no extra delay once he decides to go to a site.

We show our methods to be both theoretically sound and practical to implement using existing Internet protocols (with the addition of new client-side and server-side components). Our method also maintains compatibility for unmodified clients, although their traffic does not receive the benefit of DoS resistance. We describe a prototype implementation of our system that protects TCP connections in a manner that is transparent to client and server applications.

1.3 Organization

The paper is organized as follows. We describe our puzzle construction and distribution methods in Section 2. In Section 3 we describe how a system can be built using our D-H puzzle construction, and we discuss some extensions to our scheme in Section 5. We follow by describing our TCP-level implementation along with experimental results in Section 4. Then we discuss a few ways in which our basic results can be extended in Section 5. Finally, in Section 6 we describe related work, and we conclude in Section 7.

2. PUZZLE CONSTRUCTION

In this section we present our main Diffie-Hellman based puzzle construction scheme, which will be the construction of choice for the rest of the paper. We begin by enumerating the goals we would like our puzzle construction to meet. Next, we present our D-H based construction along with an identity-based variant. Finally, we present two other puzzle constructions that prove interesting to examine.

Lack of space forbids our including formal definitions and security proofs here; thus what is presented are construction sketches only and heuristic hardness claims. This is not to discount the importance of a formal model. On the contrary, formal definitions for puzzle hardness [22, 15] are only incipient in the literature and would naturally require extension to the outsourcing scenario as a prerequisite for security analysis. This is beyond the scope of our present investigation.

Let us introduce some notation. Let $f_k : \{0,1\}^* \rightarrow \{0,1\}^k$ be a one-way hash function whose range consists of k-bit strings. It is convenient to model f as a random oracle. The value k is a security parameter; we drop this superscript where appropriate for visual clarity. A parameter l serves to govern the hardness of the puzzle constructions we describe.

For a channel c, a timeslot τ, and a defending server, I, let $\pi_{I,c,\tau}$ denote a published and authenticated puzzle. Let $\sigma_{I,c,\tau}$ denote the corresponding solution (which we assume to be unique).

We let y_I denote the public key associated with a particular defending server I, while x_I denotes the corresponding private key; we let y and x be the respective keys of the bastion. We omit the subscript I where context makes it clear.

2.1 Goals for our scheme

Puzzle outsourcing for our purposes introduces a new set of constraints and requirements.

Recall that every timeslot and channel in our solution has only one associated puzzle. Hence, for any given timeslot the total number of puzzles is equal to the number of valid channels—perhaps on the order of thousands, according to the parameterizations we envision and describe below. In strict contrast to previous puzzle-based DoS systems, the defending server in our scheme can afford to invest fairly considerable computational resources in puzzle construction and solving. Even the computation of a modular exponentiation per puzzle is acceptable. This provides us the flexibility to introduce puzzle constructions based on public-key cryptography.

At the same time, outsourcing imposes a new set of goals for puzzle construction. We enumerate the most important of these here:

1. **Unique puzzle solutions:** The practicality of our solution depends on the ability of a defending server to precompute puzzle solutions prior to their associated timeslot, and subsequently to check their correctness via table lookup. Consequently, it is important that puzzles have unique solutions (or a very small number of correct ones).

2. **Per-channel puzzle distribution:** We want the bastion to be able to compute and disseminate puzzle information on a per-channel basis. In other words, the bastion should be able to publish information for a particular channel number c that may be used to deduce the corresponding puzzle for any defending server. (Different servers should have different puzzle *solutions*, though, so that one server's ability to enumerate its own puzzle solutions does not expose other servers to attack.)

   With this property, the bastion does not even need to know which servers it is helping to defend. This reduces the amount of information the bastion must compute and publish, and it removes the need for explicit relationships or coordination between defending servers and bastions.

3. **Per-channel puzzle solution:** Another desirable property is for the work done by a client to apply on a per-channel basis, rather than a per-puzzle basis. In particular, we would like a client that has solved a puzzle for a particular channel to be able to efficiently compute the token for the same channel number on *any* server.

   As we have already noted, this does not mean that tokens should be identical across servers—only that there should be considerable overlap in the brute-force computation needed to solve the puzzle for a given channel-number across servers. In particular, it is not desirable for one server to be able to use its shortcut to compute the tokens associated with another server, as this would result in a diffusion of trust across all participating servers rather than in the bastion alone.

   The per-channel puzzle solution property is useful because it allows a client to begin solving puzzles before deciding which server to visit.

4. **Random-beacon property:** Sometimes it is possible to achieve a property even stronger than per-channel puzzle distribution. Ideally, puzzles might not require explicit calculation and publication by a bastion. Instead they might be derived from the emissions of a *random beacon*.

   We use the term random beacon to refer to a data source that is: (1) unpredictable, i.e., dependent on a fresh source of randomness; (2) highly robust, i.e., not subject to manipulation or disruption; and (3) easily accessible on the Internet. A puzzle construction based on a random beacon would eliminate the need for explicit bastion services. (Apart from the architectural advantages, this could have the benefit in some circumstances of eliminating any point of legal liability for reliable puzzle distribution.) Hashes of financial-market data or even of Internet news sources, which both can be obtained from numerous locations, would be candidate random beacons.

   Surprisingly, under this construction not only would the bastion (random beacon) not have to know what defending servers were relying on its services, but in fact it wouldn't even need to know its data was being used to construct puzzles!

5. **Identity-based key distribution:** When puzzles are based on the public key of a defending server, the public key itself must be distributed via a robust directory. A desirable alternative is identity-based distribution, wherein the public-key of a particular defending server can be derived from the server name and a master key known to all defending servers. This is closely analogous to the well-known primitive of identity-based encryption [10].

6. **Forward security:** A final desirable property is forward security. Specifically, that time-limited passive

compromise of a bastion should not undermine the DoS protection it confers.

2.2 A D-H based construction

We now describe a puzzle construction based on Diffie-Hellman key agreement [14]. It has all of the properties above except the random-beacon property (i.e., it has properties 1,2,3,5 and 6).

Let G be a group of (prime) order q. Let g be a published generator for the group and l be a parameter denoting the hardness of puzzles for this construction. (As explained below, we require a strong, generic-group assumption on G.)

We propose a simple solution in which the bastion selects a random integer $r_{c,\tau} \in_R Z_q$ and a second random integer $a_{c,\tau} \in_R [r_{c,\tau}, (r_{c,\tau} + l) \bmod q]$. (Recall that l is the hardness parameter for the puzzle.) Let f' in this case be a one-way permutation on Z_q, and let $g_{c,\tau} = g^{f'(a_{c,\tau})}$.

The intuition is as follows. The value $g_{c,\tau}$ may be viewed as an ephemeral Diffie-Hellman public key. A puzzle solution for defending server I is the D-H key that derives from its public key $y_I = g^{x_I}$ (x_I is the secret key) and the ephemeral key $g_{c,\tau}$. Solving a puzzle means solving the associated D-H problem. To render the problem tractable via brute force, the bastion specifies a small range $[r_{c,\tau}, (r_{c,\tau} + l) \bmod q]$ of possible seed values for its ephemeral key. In other words, the bastion publishes $\pi_{c,\tau} = (g_{c,\tau}, r_{c,\tau})$.

For a client (or attacker) to solve the puzzle requires brute-force testing of all of the seed values. In particular, for a given candidate value a', the client tests whether $g_{c,\tau} = g^{f'(a')}$. For a particular defending server I, the solution to the puzzle is $\sigma_I = {y_I}^{f'(a_{c,\tau})}$.

Of course, a defending server can use its private key x_I as a shortcut to the solution of the puzzle. The defending server can compute $\sigma_I = {y_I}^{f'(a_{c,\tau})} = {g_{c,\tau}}^{x_I}$. In other words, it essentially computes a Diffie-Hellman key. For a defending server, solution of a puzzle essentially requires just one modular exponentiation.

On average, puzzle solution by a client (or attacker) requires $l/2$ modular exponentiations over G.

Since puzzle hardness needs to be precisely characterized, we believe that any concrete computational hardness claim would have to depend on a random-oracle assumption on f' and also a generic-model assumption for the underlying group G [31]. Thus it is important to choose G appropriately. (Several common types of algebraic groups are believed to have the ideal properties associated with the generic model, e.g., most elliptic curves and the order-q subgroup G of the multiplicative group Z_p^*, where $p = kq + 1$ for small k [31].)

Remark on application of f': Applying f' in the computation of ephemeral key $g_{c,\tau} = g^{f'(a_{c,\tau})}$ is a requirement to break algebraic structure among seed-to-key mappings. If we chose $g_{c,\tau} = g^{a_{c,\tau}}$, for example, then it would be possible to cycle through candidate seed values by computing $g^{r_{c,\tau}}$ and repeatedly multiplying by g.

2.2.1 *Identity-based public keys*

Here we briefly and informally sketch a technique for distributing the public keys $\{y_I\}$ of defending servers in an identity-based manner. In other words, we show how y_I can be derived from a string representing the identity I (e.g., a domain name) and a master private key. A trusted dealer can distribute individual private keys to servers using the master key. This technique can be viewed as a variant of our D-H based construction.

Employing the notation of Boneh and Franklin [10] (with which we assume familiarity here for the sake of brevity), let $\hat{e} : G \times G \rightarrow G'$ be an *admissible* bilinear mapping in the sense defined in [10] where G and G' are two groups of large prime order q. For G suitably chosen as a subgroup of the additive group of points of an elliptic curve E/F_p for prime p, $\hat{e}$ may be constructed using the Weil pairing. Recall that when the system is correctly parameterized it is believed that the Bilinear (Computational) Diffie-Hellman (BCDH) Assumption holds. This is an essential hardness property for our proposal here. Roughly stated, given $P \in_R G$ and points aP, bP, and cP for $a, b, c \in_R Z_q$, it is hard to compute $\hat{e}(P, P)^{abc}$.

Let x' be the private key of the trusted dealer, and let $y' = x'g$ be an associated public key. Finally, let $d : \{0, 1\}^* \rightarrow G$ be a one-way function mapping identifier strings to group elements in G.

In this scheme, the public key of defending server with identifying string I is computed simply as $y_I = d(I)$. The associated private key, computable by the trusted dealer, is $x'y_I$.

As before, we let $a_{c,\tau} \in_R [r_{c,\tau}, (r_{c,\tau} + l) \bmod q]$. The ephemeral key computed by the bastion assumes the form $g_{c,\tau} = f'(a_{c,\tau})g$. The bastion publishes $\pi_{c,\tau} = (g_{c,\tau}, r_{c,\tau})$, just as it does in the D-H puzzle.

The difference for the identity-based variant lies in the form of the puzzle solution. This is defined here to be $\sigma_{I,c,\tau} = \hat{e}(y_I, g)^{x' f'(a_{c,\tau})}$. (This solution may be hashed for compactness.) After solving for $a_{c,\tau}$, a client may compute this as $\hat{e}(y_I, y')^{f'(a_{c,\tau})}$. The defending server may use its knowledge of x_I as a shortcut. In particular, $\sigma_{c,\tau}^{(I)} = \hat{e}(x_I, g_{c,\tau}) = \hat{e}(y_I, g)^{x' f'(a_{c,\tau})}$.

By analogy with our D-H construction, the work for brute-force solution here is on average $l/2$ multiplications over the elliptic-curve based group G.

2.3 Other Schemes

We now discuss two other interesting puzzle constructions. The first is a hash-function-inversion construction. This construction is worth examining since its basic methods are closest to previous work on client puzzles. However, the construction does not meet properties 2, 3, 4, or 5. Its most serious limitation is that it does not meet property 2. Therefore, the bastion must compute a set of puzzles for each participating server.

The other construction we present is based upon time-lock puzzles. The most interesting property of this construction is that it meets property 4 in that puzzles challenges can be made from a random beacon. However, it does not meet property 3, so the client must compute puzzle solutions particular to the server it is contacting.

2.3.1 *Hash-function-inversion puzzle construction*

It is possible to perform outsourcing by means of partial hash-function inversion problems like those employed in previous puzzle-based anti-DoS schemes (e.g. [2, 23]).[1] Let $\sigma_{c,\tau}$

[1] A related inversion-based puzzle construction is employed in [18]. In general, this construction does not have a unique solution for a given puzzle, so it cannot be used conveniently for our purposes, as explained below.

be the j-bit secret key for $j > l$. A puzzle is computed as $f(\sigma_{c,\tau})$. To calibrate the hardness of the problem so as to require 2^{l-1} hash-function computations on average, all but l bits of $\sigma_{c,\tau}$ are revealed. For example, a puzzle might take the form $\pi_{c,\tau} = (f(\sigma_{c,\tau}), \sigma'_{c,\tau})$, where $\sigma'_{c,\tau}$ consists of all but the first l bits of $\sigma_{c,\tau}$.

To outsource the construction of such puzzles, we let x_I be shared between the defending server and bastion. (The secret x_I might be computed as a function of y and y_I via D-H key agreement.) We let $\sigma_{I,c,\tau} = f(c, \tau, x_I)$. With this approach, the defending server can quickly compute the set of solutions to puzzles for a given timeslot τ without communicating with the bastion.

2.3.2 Time-lock puzzle construction

We now propose a puzzle construction that has properties 1, 2, 4, and 6 above. It achieves the random-beacon property, but it lacks the per-channel puzzle solution property. Thus, this solution requires explicit distribution of public keys for defending servers, and a client cannot start solving puzzles prior to determining which server to access.

This construction is a simple adaptation of the time-lock puzzle scheme proposed by Rivest, Shamir, and Wagner [29]. A public key y_I consists of an n-bit RSA modulus N_I. (See [29] for discussion of restrictions on the choice of N.)

In the original RSW construction, a random value $a \in_R Z_n$ serves as a basis for the puzzle. Solving a puzzle requires the computation of a secret value $b = a^{2^l} \bmod n$. Here, b serves essentially as a key to the puzzle. The parameter l governs the hardness of the puzzle; in particular, a solver must perform l modular squarings in order to compute b and "unlock" the puzzle.

Knowledge of the factorization of n provides a shortcut to compute the secret b. For large l, computation of $e = 2^l \bmod \phi(n)$ and then $a^e \bmod n$ is much faster than brute-force squaring.

As explained above, the original RSW construction aims at creating a kind of digital time capsule—a cryptogram solvable only in the distant future thanks to advances in computing power. RSW propose that the puzzle constructor determine how hard the puzzle should be, use the shortcut in order to create an encryption key associated with the puzzle, and then erase all data associated with the shortcut, sealing the time capsule.

The main goal in the RSW design was to render the solution process difficult to parallelize, so that the ability to unlock the puzzle would truly depend upon raw advances in computing power. This property is achieved thanks to the sequential nature of the modular squarings required for the solution. By contrast, a puzzle based on hash-function inversion would not achieve this goal, since it could be divided among many different computing devices.

We exploit an altogether different property of the RSW construction (one probably not explicitly designed by its inventors). We observe that a time-lock puzzle may be derived simply from a random string (used to derive a) and an RSA modulus. *This means no explicit computation by the bastion is required to create a valid time-lock puzzle.* Our D-H solution above, on the other hand, requires the bastion to compute an ephemeral D-H key, and our hash-function-inversion puzzle scheme requires the hashing of a secret value.

Given this observation, the puzzle construction is straightforward. Let r_τ be a suitably long random string emitted by a random beacon in timestep τ (say, $n + k$ bits in length for security parameter $k \approx 128$). We let $r_{I,c,\tau} = f_{n+k}(I, c, \tau, r_\tau)$. We then compute $\pi_{I,c,\tau} = a_{c,\tau} = r_{I,c,\tau} \bmod N_I$.

The solution $\sigma_{I,c,\tau}$ to this puzzle is just $(a_{c,\tau})^{2^l} \bmod N_I$. A client (or attacker) must compute this by repeated squarings. Yet the defending server can compute it quickly using its shortcut.

Note that a single random value can be used to compute puzzles for an arbitrarily large number of channels. The security parameter l may be set by a defending server as desired. For the defending server, the work to solve a puzzle only requires a modular reduction (whose size is parameterized by l) and an RSA exponentiation. For a client (or attacker), solving the puzzle requires l modular squarings.

3. SYSTEM DESCRIPTION

In this section we describe how a system using our puzzle constructions operates in practice, and we analyze the effectiveness of our scheme. To be concrete, we use the D-H puzzle construction (but without the identity-based variant). First we describe the system parameters and operation. Then we give a practical example where we consider parameter values that might be used in practice.

In our scheme each server will have n virtual channels. The solutions to channels will be valid for a time period of length t. Typically, the period will be coarse grained so that t will be on the order of several minutes. We use T_i to denote the i-th time period.

At the beginning of the period T_i, the bastion will publish puzzle challenges having solutions that will be valid during T_{i+1}. The clients solve the puzzles distributed at the beginning of T_i during the rest of T_i and use these solutions during T_{i+1}. The server will correspondingly populate its token list for time period T_{i+1} during T_i.

For simplicity we will assume that all client machines have the same processing power to devote to puzzle solving and we view an attacker as a compromised client machine. We let s denote the average number of puzzles a client can solve during a period. The puzzle difficulty (determined by the range of possible puzzle solutions) will be set low enough that every client machine will be guaranteed to solve at least one puzzle. To ensure this we need to have $s \geq 2$.[2] During each cycle, each client will choose a random channel for which to solve a puzzle.

When a client initially contacts a specified server, it follows these steps. First the client obtains the public key for the server (for the denial-of-service system). Then the client adapts the solutions it has computed to this public key. The extra amount of computation to customize a solution for a particular public key—and thus server—is just one exponentiation. The token corresponding to the solution for the particular server and a given channel will be attached to requests made by the client.

If the server is not under attack, it will just ignore the tokens and operate normally. Yet suppose now that an attacker who controls A attacker machines begins a Denial-

[2] In reality, some machines will have more processing power to devote to puzzle solving than others. The choice of parameters will need to strike a balance between accommodating slower legitimate clients and making the puzzles difficult enough to defend against attackers.

of-Service attack. Assuming the attacker machines are well-coordinated, the attacker will be able to solve $A \cdot s$ puzzles per time period on average. If the attacker focuses an attack on time period T_{i+1} by solving puzzles for T_{i+1} over two time periods T_i and T_{i+1}, then at one point the attacker can get $2A \cdot s$ solutions.

One strategy for the attacker is to request as many resources as possible using its legitimate channels. Under this attack, the server will need to have a policy for how it divides resources among channels. For example, if the TCP layer is being protected, the server might limit the rate of SYN packets processed per channel. Observe that our abstract conception of the channel is the unit by which resources are allocated. Developing effective resource allocation policies, while important, is beyond the scope of this paper.

When the attacker machines are aggressive in requesting resources they can potentially collect all the resources allocated to the channels for which they have solutions. In this case a client that solves for one of these channels will not be able to obtain any resources. If an adversary focuses an attack on one particular time period it can occupy a fraction $\frac{2A \cdot s}{n}$ of the channels.

If a client makes a request on a channel that is not occupied by an adversary and the allocation policy permits the request, then the server can process the request immediately. Since the puzzles were solved in the previous period the user will not experience any delay due to the client puzzle system. The rate control mechanism, we note, is a function of the policy for allocating resources to each channel, not the difficulty of the client puzzles as in traditional schemes. Thus our scheme can change the resource allocation without adversely affecting user latency.

The attacker might decide to attack the puzzle defense mechanism itself, for instance by flooding the server with requests. If the requests have fake token solutions, then the overhead associated with our scheme is that of performing a memory lookup to check the token's validity. In contrast, other schemes require a hash computation at this step. (In our scheme, the server is also required to generate the list of tokens for each time period, but this effort is related to the number of channels rather than the number of requests that an attacker can make.)

In a different kind of flooding attack, the adversary makes repeated requests using a single valid token. In this case, the overhead associated with our scheme is that of checking the resources allocated to a particular channel, which again should be minimal.

We summarize the client and server operations as follows.

Client

- During period T_i, downloads random puzzles from the bastion service and solves them with spare computational resources.
- During time period T_{i+1}, uses the solutions that were solved during the previous period T_i.
- When initiating a request from a certain server, the client machine checks to see if the server has a public key for DoS prevention. If so, the client combines its puzzle solution and the server's public key to make a token for a particular channel on the server. The token is appended to the request.
- If the client has multiple puzzle solutions for multiple channels and one is not working on a particular server, the client may retry the request using a different token for a different channel.
- A client that has just booted up and stated solving puzzles may have to wait up to an entire time period before it has a solution that can be used. However, once the client is in the cycle of solving puzzles it will always have a valid solution.

Server

- During time period T_i, downloads all the puzzles for the channels and computes a token list from them using its private key. The list is used during the next period, T_{i+1}.
- If the system load is low and there is no DoS attack, then the server ignores the tokens and processes requests as though there were no DoS prevention system.
- During an attack the server only accepts requests that have valid tokens for solutions. The request tokens for a particular channel is quickly checked against the table of valid tokens. The amount of resources granted will be limited on a per channel basis.

3.1 An Example

To make our ideas more concrete, we present an example of how the parameters for our scheme might be chosen in practice.

The length of a time cycle, t, will typically be on the order of minutes. Larger values of t will allow the server more time to compute more tokens and thus offer more channels. Since the number of channels an attacker can occupy can be controlled by adjusting the puzzle difficulty, as t becomes larger, the portion of channels an adversary can control shrinks.

A large value of t has the disadvantage that a machine that has just joined the network will need to wait longer before it can have a valid solution. However, once the machine starts solving puzzles, it will solve them for the proceeding cycle, so there will be no delay after the second period following startup. Additionally, if all the channels for which a client has solutions are occupied by an adversary, then the client will need to wait for a full cycle before it can try new channels.

Using a large number of channels is advantageous in that the more channels there are, the less likely it will be that a legitimate client will solve a puzzle for the same channel as an adversary. In general, the number of channels a server can offer is limited by both the server's memory (for storing token lists and resource bookkeeping information) and the computational resources that the server can devote to solving. In our D-H solution, the computational resources of the server will be the limiting factor. As a rough estimate, a 2.1GHz Pentium processor was measured to be able to compute a 1024-bit DH key agreement in $3.7ms$ [11]. Assuming t is 20 minutes, if such a server was able to devote 1 minute or 5 percent of processing power for every cycle, then it could populate tokens for about $n = 16,000$ channels. (If we used the identity-based variant the time to compute a pairing would be around an order of magnitude more than the exponentiation, so we would have around an order of magnitude fewer channels. Therefore, the identity-based variant is currently not as practical, but it might become so in the near future as processing power increases.)

Using these parameters, we now want to determine the liklihood that a client will solve a puzzle that is not occupied by an adversary. If we set $s = 2$ then every client will solve at least one puzzle (since it can search the whole range of possible solutions for one puzzle) and half the clients will solve at least 2 puzzles. If a attack is made with 50 zombie machines then the attack, at its peak, will occupy $2 \times 50 \times 2 = 200$ puzzles and will occupy $\frac{200}{16{,}000} \times 100 = 1.25$ percent of the channels. The chances of a legitimate client not having any solutions unoccupied channels is at most $(.5 \times .0125 + .5 \times .0125^2) \times 100 \sim .625$ percent.

3.2 Projections

One important feature of our client puzzle scheme is that its ability to handle a large number of adversaries will increase as the CPU power of machines increases. As we have discussed above, the limiting resource in our scheme is the number of channels for which a server is able to compute tokens.

When the cost of public key operations decreases, a typical server will be able to support more tokens, and the cost of solving a puzzle can be adjusted such that the difficulty for the adversary stays about the same over time. In this way, technological advances will allow our scheme to efficiently handle more adversaries. Alternatively, some of the server's added power can be sifted to decreasing the initial delay, t, when a client first starts solving puzzles for the system.

Our future outlook contrasts sharply with those of traditional client puzzle schemes [23, 6, 13], which derive no obvious benefit from increases in technology (other than the server having more resources to handle requests).

4. IMPLEMENTATION

We have constructed a functioning prototype implementation of our design as it is described in section 3. The implementation consists of a suite of programs that run on the Linux operating system. They use the GNU MP Bignum library [28] for multiple precision arithmetic and the Netfilter framework [27] for network packet mangling.

Our system protects against attacks at the TCP level by regulating the rate at which new TCP connections may be established. To accomplish this, each client inserts tokens derived from its puzzle solutions into an option field of the TCP SYN packet (the first packet sent in the connection establishment process). Servers check for the presence of a valid token and use the token to separate connections into channels for rate limiting. Each channel will only accept one new connection every n seconds, where n is set by the server operator. This policy is appropriate for protecting services that use a roughly constant amount of resources for the duration of each connection.

The first program in our suite is the bastion, which creates new sets of puzzles at a regular interval. The number of puzzles in each generation, their hardness, and the time between new generations are configured by the bastion operator. Our bastion writes a set of puzzle files that are distributed by a normal web server using HTTP. We chose this design because it can be easily scaled to serve large numbers of clients by using multiple web servers or existing high-availability content distribution schemes. In each time period, the bastion creates a separate file for each puzzle, so clients only need to download the puzzle they have selected to solve, as well as a digest file containing all the puzzles, so servers only need to make one HTTP request to retrieve the entire set of puzzles.

The next program is a packet-tagging application that runs on client machines. It runs two threads: a puzzle solver and a packet rewriter. The puzzle solver waits for the bastion to post a new puzzle generation then randomly selects a puzzle and computes its solution. The packet rewriter tags outgoing SYN packets with tokens that prove the client has solved a puzzle.

The client processes packets with the Netfilter ip_queue library, which allows it to run entirely in user space. When the client detects an outgoing SYN packet, it appends a 20-byte option to the TCP header. The option consists of two tokens computed from puzzle solutions and the server's public key, along with the index of each solved puzzle in the bastion's puzzle set. We use two tokens – one solution from each of the previous two generations – to ensure that the server will accept the connection even if it has switched to a new generation somewhat sooner or later than the client. The tokens consist of the first 48 bits of the puzzle solutions. Their size is sufficiently large to prevent guessing of tokens during the time period when each puzzle is valid, yet short enough to fit in the TCP header.

Finally, we have a pair of applications that run on each server. They consist of a user space program that precomputes puzzle solutions and a kernel module that filters incoming packets. The server's user space program monitors the bastion for a new generation of puzzles and retrieves the complete set of puzzles when it is available. Then it precomputes the solution to each puzzle using the server's private key. When a subsequent generation of puzzles is posted by the bastion, the user space application transfers the previous set of solutions to the kernel module, which begins requiring that clients send solutions to puzzles from this generation.

We implemented server side packet filtering as a kernel module for speed and robustness. The module receives incoming IP packets using a hook into the Netfilter framework. We receive each packet immediately after the network subsystem has routed the packet and determined that it is destined for the local machine, and before the packet reaches higher-level protocol subsystems like TCP. If a packet is a SYN, the module begins to filter it by scanning the header for our option field and extracting the tokens and their indexes. Each token is validated by comparing it to the entry in the table of precomputed tokens corresponding to the supplied index. If either token matches, its index becomes the number of the connection's channel, and the rate limiting mechanism is applied to determine whether the connection will be accepted. Packets that exceed the rate limit or have bad tokens are immediately dropped.

4.1 Experiment

As stated before, a potential pitfall of Denial-of-Service prevention mechanisms is that they themselves will become the targets of DoS attacks. In puzzle-based solutions, if the overhead of checking puzzle solutions is too great, an attacker can overwhelm the server with a flood of packets containing bad solutions. To see how well our implementation fared against such an attack, we performed tests comparing it to two related anti-DoS mechanisms: conventional client hash puzzles and Linux's syncookies.

In our experiment we measured the load on a test server that was the target of TCP SYN flood attacks of varying intensity. The server was an 866MHz Pentium III running Redhat Linux 9.2 (kernel version 2.4.20-31.9). It was connected to three attacker machines via a 100-megabit Ethernet switch. The attack strength was modulated by employing combinations of attackers with different CPU power. Each SYN packet was tagged with an invalid puzzle token.

Our mechanism requires processor time to precompute the puzzle solutions for each generation, but exactly how much time is required depends on the puzzle parameters set by the bastion. To account for this, we measured our system in two configurations: a scenario where the server needed to calculate its tokens for 10,000 channels over a time period of 20 minutes, and a baseline configuration specially compiled to disable any token calculations. (The latter scenario rejects all TCP connections, so it is only useful for benchmarking.)

To determine system load, we counted how many loop iterations per second were performed by a process set to the lowest scheduling priority, both when the system was idle and during the attacks. In most scenarios we took the average load over a three minute period. However, to account for the uneven CPU load during token calculation in our solution, we took the average over the entire 20 minute time period when tokens were being computed.

To simulate a conventional (non-outsourced) client puzzle mechanism, we modified our kernel module to replace the puzzle verification code with a SHA-1 hash computation on 56-bytes of arbitrary data. After this computation the module drops the packet. To test syncookies we performed no filtering of our own and allowed Linux to send an ACK packet containing a cookie in response to each SYN.

The results of our experiment are plotted in Figure 1 together with a linear regression for each series. At almost all rates of attack, our solution outperformed both the SHA-based puzzles and syncookies, which had nearly equal performance. [3]

We found syncookies contributed an average of 1% load for every 541 packets per second. For the same cost, the SHA-based puzzle mechanism processed about 530 packets per second, and our method processed about 1014 packets per second. However, precomputing puzzle solutions for our scheme added a constant load of about 2.5%, regardless of attack strength. Extrapolating from this data, our scheme (with precomputing) can withstand approximately 87% more attack packets per second than SHA-based puzzles before reaching full system load, and 83% more than syncookies.

5. EXTENSIONS

5.1 Flexible number of channels

To this point we have assumed that all servers will use the same number of channels. In reality we would like to give some more flexibility to the servers. Some servers might want to tradeoff more processing time in order to provide more channels and thus handle more attackers. The primary challenge is to allow this, but in such a way that our D-H construction still has the property that a solution can be applied to any server.

We can do this in the following manner. Suppose that the maximum number of channels that a server might solve is n and that the bastion publishes n puzzles as before. We will refer to these as the primary puzzles. Clients will randomly choose puzzles to solve from among this group. Now suppose we want to allow for a server to have $\frac{n}{d}$ solutions for some d that divides n. The bastion can then create $\frac{n}{d}$ new puzzles except that instead of giving a range hint for the solutions it will encrypt these secondary puzzle solutions with the solutions of the primary puzzles. For example, if y_j denotes the solution to puzzle j in the secondary set and z_i the solution to puzzle i in the primary set then we would produce d encryption of y_j with the keys $z_{d\times j}, \ldots, z_{d\times(j+1)-1}$. Using this technique the solutions for primary channels can effectively be combined to allow for a server to have a lower number of channels.

5.2 Challenges in IP-Level Deployment

Although our implementation applies clients puzzles at the TCP level, regulating the creation of new TCP connections, the same method could be applied at other levels of the protocol stack, including the IP level. This is not true of previous puzzle-based approaches: since their puzzle solutions are more expensive to verify, a server or router could not afford to perform a puzzle verification for each IP packet. Our approach, by contrast, would require only a table lookup per packet, and so would be feasible at the IP level.

The biggest challenge we face in deploying our method at the IP level is where in the IP packet to put the token (i.e, the puzzle solution). There aren't enough unused bits in the IP header, so the logical way to attach the token is to make it an IP header option. (This approach is feasible in both IPv4 and IPv6, but it is a bit more natural in IPv6.) A header option will be ignored by routers that do not understand it; but any router or end host will be able to extract it and check it against the list of acceptable tokens.

To make this feasible for a high-capacity site, the extraction and checking of the token would have to be included in the fast-path mechanism of a router. Whether this is feasible depends on the details of how the router is designed. Space does not permit us to delve deeply into this issue, except to say that it appears to be possible on some routers but difficult on others. We leave the construction of a high-speed IP-level implementation for future work.

5.3 Eavesdropping attacks

As stated in the introduction we use an attack model where we assume that eavesdropping on the Internet is difficult for typical DoS attackers. However, it is still useful to consider what happens if eavesdropping occurs, in what situations it might occur, and measures that can be used by a client to prevent being eavesdropped upon.

If an attacker is able to eavesdrop on packets sent by a client to the server under attack, then he effectively converts the client into a drone that solves puzzles for him. This will have two repercussions. First, the attacker will be able to get another channel and consume more resources on the system as a whole. Second, the attacker will occupy the same channel as the client from which he steals tokens and that client will likely be shut out of that channel. Therefore,

[3] It may seem curious that the SHA-based puzzles and syncookies follow nearly identical load profiles. The reason is that the dominant cost of syncookies is also a SHA-1 hash computation for every packet.

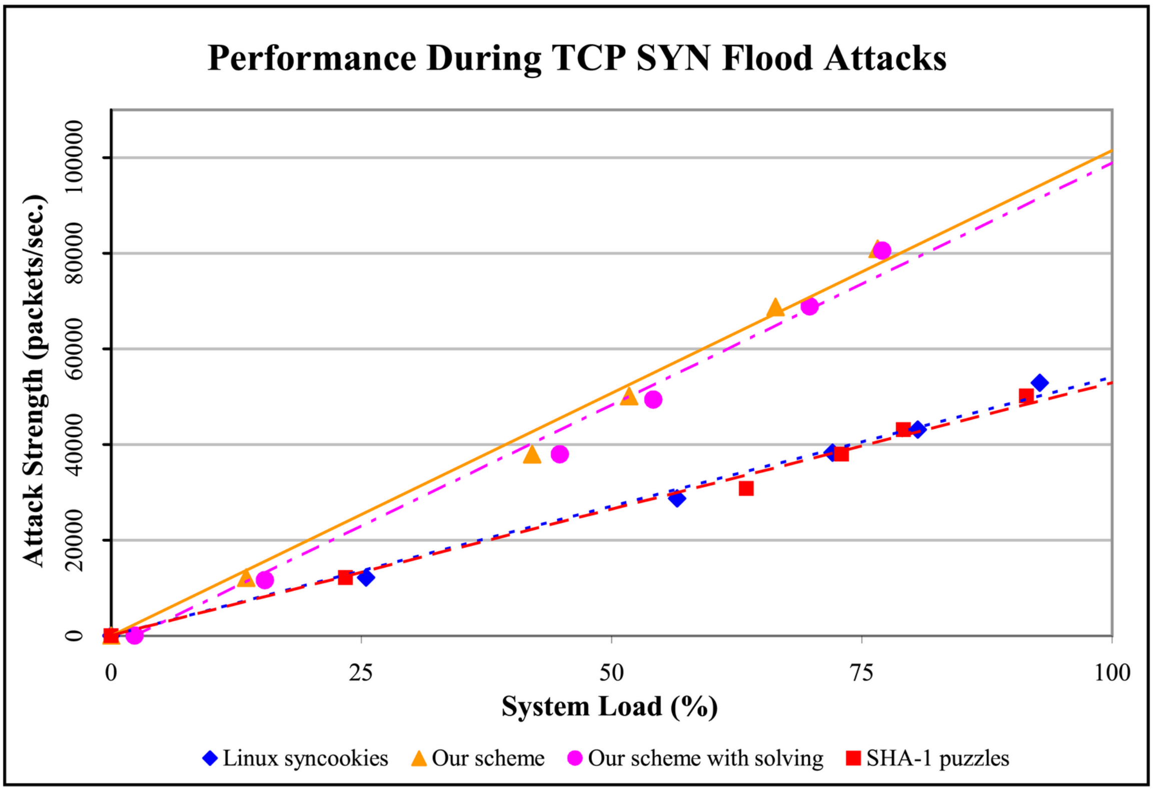

Figure 1: Graph showing the number of attack packets per second that yield a given system load using syncookies, traditional hash puzzles, and our approach.

there is a special incentive for clients not to have their tokens eavesdropped upon.

Since core routers on the Internet are difficult to compromise, the the most likely source for eavesdropping attacks are on the edge of the Internet such as a local LAN. If a client suspects that his packets are being eavesdropped upon, then it could send them securely to some part of the net that it believes to be uncorrupted. One way of doing this is to tunnel packets through IPsec [32]. We do not recommend for the server itself (or a nearby router) to act as an endpoint for such a tunnel, as the IPsec protocol could become a DoS vulnerability itself.

6. RELATED WORK

Our proposal in this paper focuses on the use of puzzles as a countermeasure to DoS attack. Dwork and Naor [16] were the first to propose puzzles for this purpose – in particular, for mitigating spam. Briefly stated, successful delivery of a piece of e-mail in their scheme requires that the sender attach a valid puzzle solution. A would-be spammer therefore faces the deterrent of a large and expensive amount of computation.

Since computational time costs money (directly or indirectly), the Dwork and Naor scheme may be thought of as akin to a micropayment system for postage. Back [7] independently devised and implemented a similar system known as Hash Cash.

The Dwork-Naor and Back systems permit pre-computation of puzzles, namely the solution of puzzles at a time arbitrarily antedating the sending of the e-mail they are associated with. This may be effective against spam, but not against the common form of DoS in which an attacker seeks to disable a server by overwhelming its resources during some restricted period of time. This common real-world DoS attack is often referred to as a *flooding attack*. It is our main focus in this paper.

Juels and Brainard [23] addressed the problem of puzzle precomputation permitting flooding with an idea they termed “client puzzles”; these are puzzles based on session-specific parameters, that can be applied to interactive protocols like TCP and SSL. Aura, Nikander, and Leiwo [6] propose variants aimed specifically at DoS attacks against authentication protocols. Dean and Stubblefield [13] focus on the application of client puzzles to SSL (or TLS), and investigate surrounding deployment issues. Wang and Reiter [36] also consider puzzle deployment for DoS protection in authentication. They devise a system in which clients bid for resources by solving puzzles of appropriate difficulty.

More recently, researchers have proposed a few variants on basic puzzle constructions. Abadi et al. [1] describe a new puzzle construction aiming at a levelling effect among computational platforms (i.e., at permitting more equal resource allocation among fast and slow machines). The puzzles they propose rely primarily on the resource of high speed memory, which tends to be more equally distributed among computing platforms than raw computational power. Dwork et al. [15] propose some improved constructions in follow-up work. Finally, CAPTCHAs [35] are a kind of puzzle that depend upon human work, rather than machine computation, for their solution. All of these puzzle variants may be adapted to our proposal in this paper.

We omit discussion here of many cryptographic and other uses of puzzles apart from combatting DoS, e.g., [17, 20, 22, 26].

6.1 Approaches to IP-layer DoS

Puzzles represent only one approach to DoS mitigation, and they have previously seen use mainly at the application or session-establishment level, rather than at lower protocol levels. As explained above, a goal of our proposal is to provide techniques efficient enough to be deployed to help low protocol layers, such as TCP, as our prototype demonstrates – or even IP, in principle. We discuss some of the existing techniques for IP-layer protection here.

One of the best known approaches to addressing IP-layer attacks is referred to as *traceback*. This involves the supplementation of packet data to permit tracing of the origins of an attack [3, 9, 12, 30, 33]. Pushback [25] and Path Identification (Pi) [37] are related IP-level approaches to DoS. They facilitate gathering of forensic data, but suffer from the need for modifications to the routing infrastructure. Anomaly detection [8, 21] is another actively researched approach to IP-level DoS that involves classification and suppression of suspicious network traffic.

A very practical approach to attacks against certain protocols (and used in real-world systems to protect the TCP SYN protocol) is known as a *syncookie*. In order to validate the claimed IP address of a client, a server transmits a (cryptographically computed) cookie to the address. The client must transmit this cookie to the server in order to have its service request completed. Thus, while not aimed at IP-layer DoS, syncookies exploit low-level network services to achieve their protection.

An important emerging thread of research on DoS that underlies our work involves redirection of potentially hostile traffic to robust loci capable of withstanding attack and providing filtering services, as in Stone [34], Andersen [4], and Keromytis et al. [24]. Recently Adkins, Lakshminarayanan, Perrig, and Stoica [2] show how to combine this approach with puzzles; among other ideas, they advocate leveraging the (proposed) Internet Indirection Infrastructure (i3) in such a way that a challenge puzzle is issued for each connection request. Our proposal is similar in flavor, but more lightweight and consequently coarser in nature. A key difference is that we advocate outsourcing from the defending server only the process of puzzle distribution, rather than broad management of incoming traffic.

In this respect, our proposal is similar to that of Anderson, Roscoe, and Wetherall [5]. They propose that a client use a *token* in order to validate a path to a server; this token serves as a packet-level nonce employable for purposes of filtering by "verification points." A token in the ARW approach serves essentially the same function as a puzzle solution in our own. The security model is similar as well: Anderson et al. assume that adversaries do not eavesdrop extensively on network links. A key difference is the way in which tokens are distributed. ARW propose incremental deployment of an infrastructure of "Request-to-Send" (RTS) servers (and do not detail the critical policy question of how transmitters are authorized to obtain tokens from RTS servers). Bastions in our proposal are analogous to RTS servers. Indeed, our proposal may be viewed as a more practical alternative to RTS servers: Bastions dispose of the need both for an infrastructure of actively intercommunicating servers and for explicit policies about token distribution.

Gligor [19] also considers the problem of the overhead of conventional client puzzle schemes and proposes an outsourcing scheme. However, his scheme relies on a third party that is positioned to verify the source IP address of the requester. We do not suppose the existence of such a party.

7. CONCLUSION

We have examined the problem of defending a server against Denial-of-Service attacks using a new technique based on client puzzles. We observe that since puzzle distribution itself can be subject to attack, any viable system must have a robust puzzle distribution mechanism. We developed a new model for puzzle distribution using a robust service that we call a bastion. The bastion distributes puzzles, and solutions to the puzzles allow clients to access communication channels. Within this model we develop different cryptographic techniques for puzzle dispersement. Our primary method, the D-H puzzle construction, has the advantages that the bastion does not need to be aware of the server's using the system and that solutions to puzzles can be computed offline, resulting in minimal user delay. Finally, we implemented a prototype of our system that works on today's Internet and experimentally demonstrated the advantages of our solution. While our implementation was directed at TCP, our hope is that future work might even demonstrate our techniques efficient enough to work for lower-layer protocols such as IP.

8. REFERENCES

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. In *NDSS '03*, pages 107–121. Internet Society, 2003.

[2] D. Adkins, K. Lakshminarayanan, A. Perrig, and I. Stoica. Taming IP packet flooding attacks. In *HotNets-II*. ACM Press, 2003.

[3] M. Adler. Tradeoffs in probabilistic packet marking for IP traceback. In *STOC '02*, pages 407–418. ACM Press, 2002.

[4] D. G. Andersen. Mayday: Distributed filtering for Internet services. In *USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[5] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet denial-of-service with capabilities. In *HotNets-II*. ACM Press, 2003.

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

[7] A. Back. Hashcash - a denial-of-service countermeasure, 2002. Original system developed in 1997. Manuscript. Referenced 2004 at http://www.hashcash.org/hashcash.pdf.

[8] P. Barford, J. Kline, D. Plonka, and A. Ron. A signal analysis of network traffic anomalies. In *Internet Measurement Workshop*, 2002.

[9] S. Bellovin, M. Leech, and T. Taylor. ICMP traceback messages, 2003. Internet Draft.

[10] D. Boneh and M. Franklin. Identity based encryption from the Weil pairing. *SIAM J. of Computing*, 32(3):586–615, 2003.

[11] Wei Dai. Crypto 5.1 benchmarks. Web site at http://www.eskimo.com/ weidai/benchmarks.html.

[12] D. Dean, M. Franklin, and A. Stubblefield. An algebraic approach to IP traceback. *Information and System Security*, 5(2):99–137, 2002.

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

[14] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[15] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In D. Boneh, editor, *CRYPTO '03*, pages 426–444. Springer-Verlag, 2003.

[16] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *CRYPTO '92*, pages 139–147. Springer-Verlag, 1992.

[17] M.K. Franklin and D. Malkhi. Auditable metering with lightweight security. In R. Hirschfeld, editor, *Financial Cryptography '97*, pages 151–160. Springer-Verlag, 1997.

[18] E. Gabber, M. Jakobsson, Y. Matias, and A. Mayer. Curbing junk e-mail via secure classification. In R. Hirschfeld, editor, *Financial Cryptography '98*. Springer-Verlag, 1998.

[19] Virgil D. Gligor. Guaranteeing access in spite of service-flooding attacks. In *Security Protocols Workshop*, 2003.

[20] D. Goldschlag and S. Stubblebine. Publicly verifiable lotteries: Applications of delaying functions. In R. Hirschfeld, editor, *Financial Cryptography '98*. Springer-Verlag, 1998.

[21] A. Hussain, J. Heidemann, and C. Papdopolous. A framework for classifying denial-of-service attacks. In *ACM SIGCOMM*, 2003.

[22] M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. In *Communications and Multimedia Security*, pages 258–272. Kluwer Academic, 1999.

[23] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the 1999 ISOC Network and Distributed System Security Symposium*, pages 151–165, 1999.

[24] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *ACM SIGCOMM*, pages 61–72. ACM Press, 2002.

[25] R. Mahajan, S.M. Bellovin, S. Floyd, J. Ioannidis, V. Paxons, and S. Shenker. Controlling high bandwidth aggregates in the network. *ACM Computer Communication Review*, 32(3):62–73, 2002.

[26] R. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(8):294–299, April 1978.

[27] The Netfilter/Iptables Project. Web site at http://www.netfilter.org.

[28] The GNU MP Project. Web site at http://www.gnu.org/software/gmp/gmp.html.

[29] R.L. Rivest, A. Shamir, and D. Wagner. Time-lock puzzles and timed-release crypto. Technical Report MIT/LCS/TR-684, MIT, 1996.

[30] S Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *ACM SIGCOMM 2000*, pages 295–306, 2000.

[31] C.-P. Schnorr and M. Jakobsson. Security of discrete log cryptosystems in the random oracle and generic model. In *The Mathematics of Public-Key Cryptography*. The Fields Institute, 1999.

[32] IP Security Protocol Charter. Web site at http://www.ietf.org/html.charters/ipsec-charter.html.

[33] D. X. Song and A. Perrig. Advanced and authenticated marking schemes for IP traceback. In *IEEE INFOCOM*, pages 878–886, 2001.

[34] R. Stone. CenterTrack: An IP overlay network for tracking DoS floods. In *USENIX Security '00*, 2000.

[35] L. von Ahn, M. Blum, N.J. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In E. Biham, editor, *Eurocrypt '03*, pages 294–311. Springer-Verlag, 2003.

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

[37] A. Yaar, A. Perrig, and D. Song. Pi: A path identification mechanism to defend against DDoS attacks. In *IEEE Symposium on Security and Privacy*, pages 93–109, 2003.