The Design and Implementation of Network Puzzles

Wu-chang Feng Ed Kaiser Wu-chi Feng Antoine Luu
Portland State University ENSEIRB
{*wuchang, edkaiser, wuchi*}*@cs.pdx.edu* *antoine.luu@free.fr*

Abstract— **Client puzzles have been proposed in a number of protocols as a mechanism for mitigating the effects of distributed denial of service (DDoS) attacks. In order to provide protection against simultaneous attacks across a wide range of applications and protocols, however, such puzzles must be placed at a layer common to all of them; the network layer. Placing puzzles at the IP layer fundamentally changes the service paradigm of the Internet, allowing any device within the network to push load back onto those it is servicing. An advantage of network layer puzzles over previous puzzle mechanisms is that they can be applied to all traffic from malicious clients, making it possible to defend against arbitrary attacks as well as making previously voluntary mechanisms mandatory. In this paper, we outline goals which must be met for puzzles to be deployed effectively at the network layer. We then describe the design, implementation, and evaluation of a system that meets these goals by supporting efficient, fine-grained control of puzzles at the network layer. In particular, we describe modifications to existing puzzle protocols that allow them to work at the network layer, a** *hint-based hash-reversal puzzle* **that allows for the generation and verification of fine-grained puzzles at line speed in the fast path of high-speed routers, and an `iptables` implementation that supports transparent deployment at arbitrary locations in the network.**

I. Introduction

The Internet currently carries an enormous amount of undesirable network communication. This is evidenced by the growing infestation of worms and viruses such as Nimda, Code Red, and SQL Slammer [1], [2], [3], reconnaissance attacks such as port scans, targeted distributed denial-of-service attacks, and spam. Client puzzles [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15] have been proposed as a mechanism for controlling such communication. With client puzzles, a server or network being protected generates a cryptographic puzzle that a client must answer correctly before it is given service. Such a mechanism gives devices the ability to selectively push back load to the source of an attack when overloaded. While the standard defense for preventing undesirable communication is to apply a binary filter to traffic, such a defense is difficult to use due to the impact of false positives and the inability to completely differentiate good traffic from bad. Client puzzles provide a complementary weapon to filtering in that they provide an analog control against traffic that may potentially be deleterious. In contrast to filtering, client puzzles also limit an attacker's ability to send bad traffic to multiple victims concurrently by consuming their computational resources.

One of the limitations of current approaches for using client puzzles is that they can be easily thwarted if an adjacent or underlying protocol does not implement them. In order to provide reasonable protection across applications, it has been argued that such a mechanism must be placed at a layer common to all Internet communication: the IP layer [15]. The design of the IP layer has been driven by the "end-to-end principle" [16], a set of guidelines that argues against putting special-case functions into common network layers. As a result, only essential functions have been placed in the network layer while all other functions have been implemented at the end-points.

Client puzzles provide an essential function that is common to all applications and should be placed in the IP layer. The observation that denial-of-service activity can happen at any layer and only needs to break one link in the end-to-end chain in order to be successful leads to the "weakest-link" argument to protocol design:

Put in the common waistline layer functions whose properties are otherwise destroyed unless implemented universally across a higher and/or lower layer.

In particular, functions such as congestion control and DoS prevention require global deployment in order to be effective. For example, TCP congestion control is thwarted by UDP flooding and DoS-resistant authentication protocols are thwarted by IP flooding. Until puzzles are placed within IP, IP will remain the weakest link.

This material is supported in part by the National Science Foundation under Grant ANI-0230960 and the generous donations of Intel Corporation. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or Intel.

Motivated by the weakest-link argument, this paper describes the design and implementation of network layer puzzles. There are two key properties of our design; a protocol which supports the issuance of puzzles at a variety of resource granularities and at any time during the lifetime of a flow, and a novel fine-grained puzzle mechanism that can support fast generation in high-speed routers.

Section II describes the design goals for supporting and deploying puzzles at the network layer. Section III describes the design of the puzzle protocol. Section IV evaluates our novel puzzle mechanism with respect to a number of other puzzle mechanisms for use in the network. Section V describes and evaluates a Linux-based `iptables` implementation that uses IP options and ICMP.

II. Goals

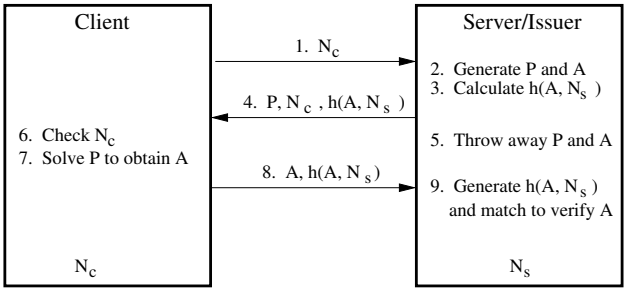
There are several important goals that must be achieved in order for client puzzles to be deployed effectively at the network layer. These goals include:

- *Flexible deployment*: The protocol must be sufficiently flexible to support puzzle issuance at arbitrary points in the network, including at end-hosts, firewalls, and routers.
- *Flexible usage*: The protocol should support the issuance of puzzles at arbitrary resource granularities such as on a per-host, per-flow, or even per-packet basis. Specifically, it must allow puzzles to be issued at any point during the lifetime of a flow.
- *Tamper-resistance*: The protocol should limit replay attacks over time and space. Puzzle answers should not be valid indefinitely and should not be usable by other clients. While the protocol should limit spoofing attacks, a specific non-goal is strong authentication between the client and issuer since the issuer may not be the end-host. This work assumes that the adversary does not lie along the path from the client to the server (i.e. the adversary cannot read or modify packets sent between the client and the server). Such an assumption is reasonable since an adversary that lies along the path can execute a more effective DoS attack using fewer resources than manipulating puzzles; the attacker could drop all packets. As a result, the system should prevent spoofing attacks only from adversaries who do not lie along the path from the client to the server.
- *Efficiency*: The protocol and implementation must be efficient in terms of memory and CPU overhead at the issuer. Specifically, puzzle generation and verification should add minimal overhead to network devices in the *worst-case* [17] to prevent the puzzle protocol from becoming an avenue for denying service. In addition, the amount of header/packet overhead should be limited to minimize the effect of reflector attacks [18].
- *Minimal application impact*: The use of the puzzle protocol should not break latency-sensitive applications such as interactive voice, streaming video, and networked games. Clients who are able and willing to solve puzzles should be able to run all of their applications seamlessly.

III. Protocol Design

Many of the above goals can be addressed via mechanisms described in a variety of previous protocols. This section describes a basic protocol developed from previous puzzle work [7], [8], [9], [10], [14], [19] and from TCP SYN cookies [20], followed by the modifications that are necessary to allow the protocol to operate at the network layer. In the remainder of this paper, *puzzle server* refers to the network device that issues the puzzles, while *puzzle client* refers to the client that solves the puzzles.

A. Basic Puzzle Protocol



Protocol Field	Description
N_c	Client nonce
N_s	Server nonce
P	Puzzle
A	Answer
$h()$	Cryptographic hash function

Fig. 1. Basic puzzle protocol

Figure 1 shows the basic protocol which supports constant-state operation at the server and client. The only state required is a set of randomly-generated, periodically-updated client nonces (N_c) and server nonces (N_s). In order to get the client to solve a puzzle, a server must echo a client nonce correctly, thus preventing spoofing attacks from third parties that are not along the path of communication. Client nonces also prevent a server from continually issuing puzzles indefinitely to a client that is no longer requesting service. Server nonces are kept secret and are used to efficiently verify answers. Since attacks on pseudo-random number generators are possible, both client and server nonces should be generated using a "true" random number generator [21], [22], [23], [24].

The protocol initially starts with a packet stream. The client attaches a client nonce (N_c) to each packet it forwards. Upon receiving a packet that triggers the puzzle

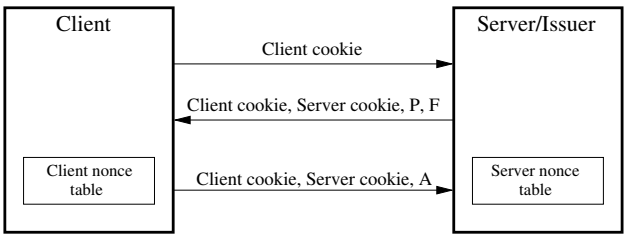
mechanism, the server generates a puzzle (P) and answer (A) as well as a cryptographic hash of the answer and server nonce ($h(A, N_s)$). The server returns the client nonce, puzzle, and hash. Generating a cryptographic hash (i.e. SHA1) of the answer with a sufficiently random nonce allows the the server to discard everything except the nonce, while retaining the ability to verify correct answers. Clients check the echoed client nonce against its set of nonces in order to verify that it is still valid before solving the puzzle. After solving the puzzle, the client attaches the answer and hash to all subsequent packets to the server. To verify answers sent by the client, all the server must do is hash the answer with the server nonce and check if the generated hash matches the one echoed by the client. If it does, the correct answer has been given and the server accepts the packet.

B. Protocol Modifications for IP

While the basic protocol has many salient features, a few issues remain to be addressed before puzzles are feasible at the IP layer. These include:

- *Efficient nonce verification*: A problem with using a set of nonces is the memory overhead of a nonce lookup. Since many network devices are memory-bound and high-speed memory is prohibitively expensive [25], reducing the number of memory accesses is critical. To support efficient nonce lookup at both the client and server, logical timestamps (TS_c and TS_s) are used to directly index into the nonce table. With them, nonce verification requires only a single memory access.
- *Strict control of answer replay between and within flows*: Solving a single puzzle should not give clients unlimited access. For example, in the case of port scanning tools (such as `scanrand`, `nmap`, and `nessus`), solving a single puzzle should not allow connections to all other ports on a host to occur without additional puzzles being solved. To address this, a flow identifier (F) can be included in the hash to bind puzzles and answers to particular packets and flows. Upon receiving an answer, the server uses the packet's flow information when verifying the answer. For example, if the server wishes to implement per-flow puzzles, F can include the connection identifier 5-tuple (source IP, destination IP, source port, destination port, protocol), thus forcing the client to solve a new puzzle for each new connection. To allow the client to know which flow to bind puzzle answers to, the flow identifier must be attached to the puzzle.
- *Strict control of answer replay over time*: Network puzzles can potentially provide routers with a mechanism for performing mandatory congestion control. In order to finely control resource usage over short periods of time, however, the server may require puzzles to expire at a much finer frequency than its nonce is changed. To support this, a puzzle expiration time (T_e) similar to those used in client authentication protocols [8], [19], [26], [27] can be added to the protocol. The puzzle expiration time enables the server to force clients to continually solve new puzzles without forcing the server to change its nonce at the same rate. The server nonce needs only be updated at a frequency that would thwart brute-force attacks on it.
- *Supporting latency sensitive applications*: Forcing a client to stop and solve a puzzle before continuing service can adversely impact the usability of interactive and streaming applications. It should be possible to issue puzzles ahead of time, allowing clients to solve them beforehand so that they can smoothly transition between two puzzle answers and continue service uninterrupted. In order to support this mode of operation, a puzzle maturity time (T_m) is included in the protocol. In steady state, the client uses a pre-calculated answer to a puzzle that has matured while calculating the solution to the next, maturing puzzle.

C. Full Puzzle Protocol



Protocol Field	Description
Client cookie	N_c, TS_c
Server cookie	$TS_s, T_m, T_e, h(A, N_s, TS_s, T_m, T_e, F)$
P	Puzzle and parameters (hints, difficulty)
F	Flow identifier
A	Answer
N_c	Client nonce
TS_c	Client timestamp
N_s	Server nonce
TS_s	Server timestamp
T_m	Puzzle maturity time
T_e	Puzzle expiry time
$h()$	Cryptographic hash function

Fig. 2. Full puzzle protocol

Figure 2 shows the final protocol with all of the protocol components. The client attaches a cookie consisting of its nonce and a timestamp. A server requiring puzzles generates a puzzle and answer along with a hash of the answer, server nonce, puzzle expiration time, puzzle maturity time, and flow identifier. The server then sends back to the client: the client cookie, puzzle and its parameters, flow identifier, and a server cookie consisting of the above hash, server timestamp, puzzle maturity and expi-

ration times. The client, upon receiving the puzzle, calculates the solution and sends back the answer along with the server cookie. Upon receipt of this message, the server uses the server timestamp to index into the server nonce table to obtain the server nonce, checks that the nonce has not expired, and verifies the answer by regenerating the hash and comparing it against what the client sent.

IV. Puzzle Mechanism Selection

While the puzzle protocol facilitates the efficient deployment of puzzles at the network layer, the puzzles themselves must be appropriately designed for use with our protocol. In this section, we examine the trade-offs when selecting a puzzle mechanism for use in the network layer. In particular, we focus on two properties: *efficiency* and *resolution*. In terms of efficiency, it must be possible to generate puzzles and verify answers on the order of microseconds to support large streams of packets from a vast number of clients (i.e. high-speed routers must be able to perform puzzle generation and verification in the fast path). In terms of resolution, it must be possible to finely control the amount of work given to a client to maintain high utilization. Puzzles that are too coarse lead to resource underutilization similar to that seen with TCP at low levels of multiplexing.

In this section, we analyze three existing puzzle mechanisms: time-lock puzzles, hash-reversal puzzles, and multiple hash-reversal puzzles. We introduce *hint-based hash-reversal puzzles* as an alternative that is best suited for the network layer and can be implemented directly in network devices. Finally, we compare the four mechanisms.

A. Time-Lock Puzzles

Time-lock puzzles are based on the notion that a client must spend a particular amount of computation time performing repeated squaring; a sequential process that forces the client to compute in a tight loop for a controllable amount of time [28]. With time-lock puzzles, the server estimates the number of squaring operations a client can perform per second (S), and the amount of time it wants a client to spend solving the puzzle (T). It calculates the number of squarings that must be performed to solve the puzzle, $t = T \times \mathcal{S}$, and forces the client to calculate $b = a^{2^t} (mod\ n)$. Time-lock puzzles are an attractive puzzle type since they provide an exact, fixed amount of work.

Time-lock puzzle generation requires two large prime numbers p and q, which take significant server resources to generate. Unfortunately this means time-lock puzzles cannot be efficiently generated on the order of microseconds.

B. Hash-Reversal Puzzles

Another puzzle approach is to force clients to reverse cryptographic hashes calculated at the server given the original random input with n bits erased [7]. In order to vary the difficulty level, n is either increased or decreased. The client performs a brute-force search on the erased bits by hashing each pattern in the space until it finds the answer. Since a single hash can be performed quickly and is compact, puzzle generation time and size are significantly less than those of time-lock puzzles. Also, many network devices have hardware support for cryptographic hashing and random number generation, making it possible to generate these puzzles at line speed.

Hash-reversal puzzles have a few disadvantages. The first is that their solution time is probabilistic in nature and is based on how lucky the client is in its search. A search could terminate after the first try or after the 2^n-th try. When applied over a large number of puzzles (as would be the case for network puzzles), the average difficulty will converge to the desired level, making this an insignificant disadvantage. A second disadvantage is that the puzzle can be parallelized by splitting the search range up amongst a number of different systems. This disadvantage is also insignificant since the same systems could be used directly in a distributed denial-of-service attack to the same effect. The only significant disadvantage is that adjacent difficulties vary by a factor of two. Solving an n bit puzzle is twice as hard as solving an $(n-1)$ bit puzzle. Due to this coarseness, it is hard to establish an appropriate hash-reversal puzzle difficulty that maximizes utilization.

C. Multiple Hash-Reversal Puzzles

Dividing the puzzle into multiple smaller hash-reversal puzzles as proposed by Juels [7] can mitigate the disadvantages of hash-reversal puzzles. The chances of being lucky on each sub-puzzle becomes small, decreasing the variance in total solution time. Furthermore, using sub-puzzles of varying difficulty allows finer control of the overall puzzle difficulty. For example, if the overall difficulty requires $(2^{10}+2^8)$ hashes worth of work, sub-puzzles of 10-bits and 8-bits could be sent to the client instead of sending either a 10-bit puzzle or an 11-bit puzzle. Figure 3 demonstrates the puzzle difficulties supported as a function of the total number of bits used across all sub-puzzles using one, three, and six sub-puzzles. The figure shows a fine resolution at low difficulties, with resolution exponentially worsening as the difficulty linearly increases.

While multiple sub-puzzles can improve difficulty resolution, it does so via a linear increase in generation time and puzzle size. In order to finely control the resolution at large difficulties, a puzzle must consist of many sub-

puzzles. To maintain fine-grained control across heavier workloads with faster client CPU speeds, the number of sub-puzzles must increase. This prevents multiple hash-reversal puzzles from being a viable puzzle mechanism.

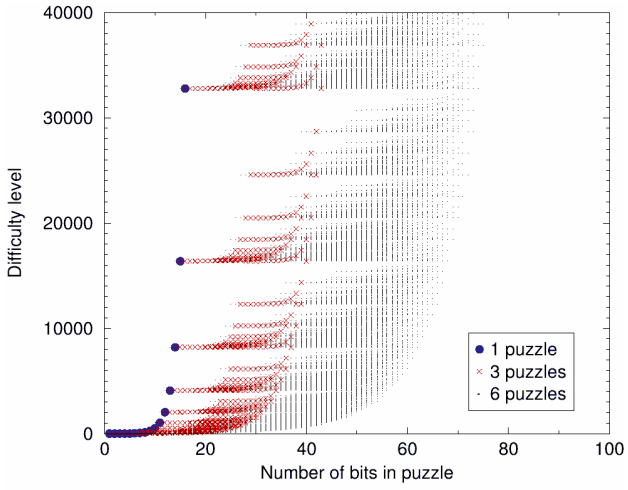


Fig. 3. Puzzle difficulties supported using single and multiple hash-reversal puzzles

D. Hint-Based Hash-Reversal Puzzles

We propose a novel mechanism for delivering fine-grained puzzles in which a single hash-reversal puzzle is given to the client along with a hint that gives the client an idea of where the answer lies. The hint is a single value that is near the answer and solves the coarseness problem of hash-reversal puzzles. To adjust the difficulty of the puzzle, the accuracy of the hint is increased or decreased. For example, suppose a randomly generated number x is used as the input to the hash $h(x)$. To generate a puzzle with $O(D)$ difficulty, the issuer passes the client the hash and a hint, $x - u(0, D)$. Where $u(0, D)$ is a randomly chosen number uniformly distributed between 0 and D. The client then starts at the hint and searches the range linearly for the answer. The number of hashes done by the client to find x varies probabilistically but the expected value is $\frac{D}{2}$.

E. Puzzle Comparison

To compare the puzzle types, Table I lists the properties of each puzzle type. Unit work (w) describes the basic operation the client must repeatedly perform to solve the puzzle and the average amount of time the operation requires on our evaluation system (an unloaded 1.8GHz Pentium 4). Range describes the range of difficulties supported by the puzzle based on n, the number of bits in the secret. The mean and maximum resolution describe the spacing between adjacent puzzle difficulties.

As the table shows, time-lock puzzles can be given at a very fine resolution all the way up to the maximum difficulty level which is bounded by a brute-force search on the server's secret nonce (given an n bit nonce, the maximum puzzle difficulty is $O(2^n)$). In contrast hash-reversal puzzles have much coarser resolution, especially at higher difficulty levels. Multiple hash-reversal puzzles can alleviate the resolution problem based on k, the number of n-bit sub-puzzles. While the derivation is out of the scope of this paper, it can be shown that the number of distinct difficulty levels is a closed function of k and n, as shown in the table. Hint-based hash-reversal puzzles have a very fine resolution comparable to that of time-lock puzzles.

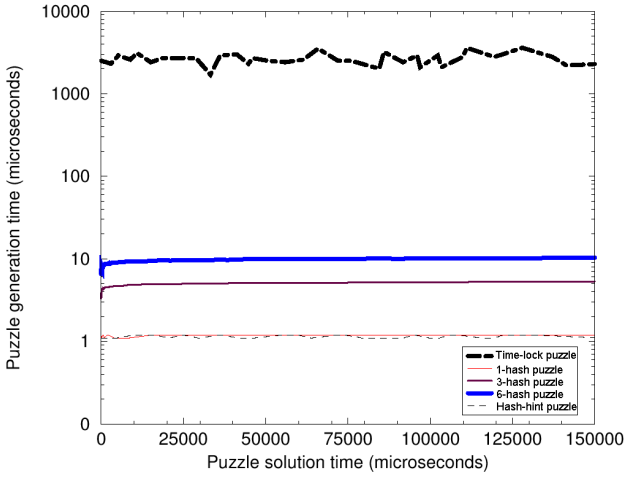


Fig. 4. Puzzle generation versus solution time

Figure 4 shows the generation time of each puzzle type as a function of the solution time across a large range of difficulty levels. Each data point represents an average of 100 different puzzles which were generated and solved on our evaluation system. As the figure shows, the generation time for time-lock puzzles is several orders of magnitude greater than that of any of the hash-reversal puzzle types.

F. Answer Verification

The answer verification mechanism is the same across all puzzle types; cookies are used to support constant-state verification of answers. Clients must present their solution with the server cookie which was attached to the puzzle. To verify correctness, the server uses the timestamp to index into the nonce table and obtain the corresponding nonce, performs a hash of the client's solution with the nonce, and checks to see if it matches the echoed server cookie. These operations are simple, allowing the server to verify puzzles very quickly. Across 1000 puzzle verifications on our evaluation system, the average time was 1.24 μs (i.e. > 800,000 per second).

Puzzle Type	Unit Work (w)	Range	Mean Resolution	Max Resolution
Time-Lock	squaring ($0.75\mu s$)	$O(2^n)$	w	w
Single Hash-Reversal	hash ($1.09\mu s$)	$w * 2^n$	$w * \frac{2^n}{n}$	$w * 2^{n-1}$
Multiple Hash-Reversal	hash ($1.09\mu s$)	$w * k * 2^n$	$\frac{w*k*2^n}{\sum_{i=0}^{k}(n-i)\binom{n}{i}}, k \leq n$ $\frac{w*k*2^n}{(k-n+1)2^n+\sum_{i=0}^{n-1}(n-i)\binom{n}{i}}, k > n$	$w * 2^{n-1}$
Hint-Based Hash-Reversal	hash ($1.09\mu s$)	$w * 2^n$	w	w

TABLE I

PUZZLE SOLUTION CHARACTERISTICS

V. IMPLEMENTATION

To demonstrate the feasibility of our protocol and puzzle algorithm, we implemented our design in Linux using `netfilter` and `iptables` [29]. This section describes the details of our implementation, provides an example deployment scenario, and evaluates the implementation.

A. Details

The implementation uses the Linux kernel modules `netfilter` and `iptables` to provide hooks and support for modifying packets in the kernel. Our system implements the protocol using two modules: a puzzle issuing firewall and a puzzle solving proxy. We found that for thin clients that do not possess the computational power required to solve the puzzles, it is possible for an administrative domain to set up a proxy machine to solve the puzzles without violating the protocol or its intentions.

There are two possible and acceptable scenarios where a proxy will become a bottleneck. The first is that the proxy is working on behalf of clients who are behaving maliciously and are being issued very difficult puzzles. In this case it is desirable that the proxy is a bottleneck since each attacker using the proxy is throttled by the cumulative difficulty of all puzzles issued to the attackers. Administrators can fix the bottleneck for legitimate users by disconnecting and repairing the machines which are creating the malicious traffic. The second scenario where a proxy will become a bottleneck is that the proxy is attempting to solve puzzles for too many clients. In this case, the administrators simply did not allocate an adequete number of proxies to handle the legitimate users.

The system uses ICMP source quench messages to deliver puzzles, and IP options to transmit client cookies and puzzle answers. Figure 5 shows how the protocol messages are attached to a packet stream.

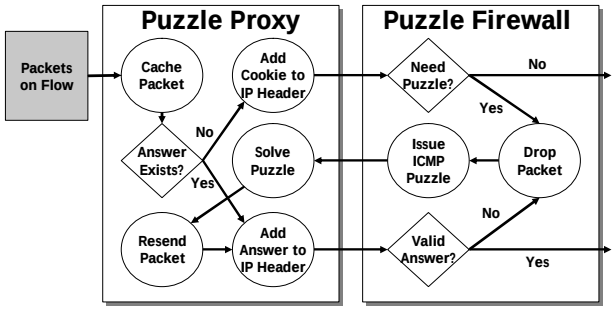


Fig. 5. Protocol messages in action

The puzzle proxy attaches the client cookie (the IP option shown in Figure 6) to all outgoing packets in a stream and caches a copy of the latest packet. Upon receiving packets from a source that requires a mandatory quench, the puzzle firewall sends a hint-based hash-reversal puzzle (the ICMP packet shown in Figure 7) back to the client. The ICMP puzzle is effectively a mandatory version of the pre-existing ICMP source quench [30], where a client demonstrates it has quenched itself by attaching correct answers to its subsequent packets. It is important to know that the puzzle difficulty is a 32-bit unsigned integer ($\text{difficulty} \in [0, 2^{32}]$) and a difficulty of 0 means that no puzzles are required. When a puzzle is received by a puzzle proxy, it verifies the echoed cookie and then solves the puzzle. After solving the puzzle, the proxy attachs the answer (the IP option shown in Figure 8) to all future packets on that flow. The proxy also resends the cached packet which triggered the puzzle. When the puzzle firewall receives a packet with an answer it checks the answer before forwarding the packet. Any time an answer is not valid (most often due to the answer expiring) the firewall drops the packet and sends a new puzzle to the client. If the network drops a puzzle, the next packet on the flow will trigger another puzzle since it will also be invalid.

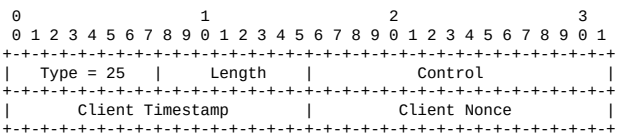


Fig. 6. Client cookie IP option

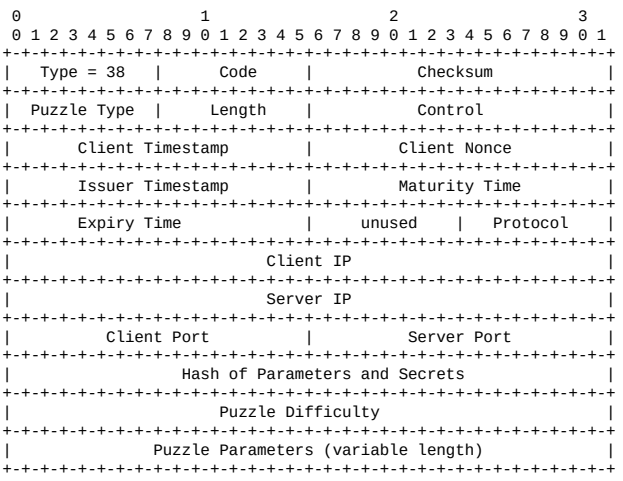


Fig. 7. ICMP puzzle

B. Deployment Scenario

To demonstrate how the modules can be used, Figure 9 shows a simple proxy-firewall setup and packet trace. The client (`ak47`) behind the proxy initiates two connections to the destination network being protected by the firewall. The first connection is to a closed port on a protected server (`mp5:2601`), while the second is to a non-existent machine (`10.0.2.123:23`). When not using network puzzles, the client would simply receive an RST segment in response to the first connection and receive no response to the second connection. However, when using network puzzles, the firewall issues a puzzle for each connection attempt. The proxy, on behalf of the client, must then solve each puzzle before the client can find out whether or not the service or machine it is seeking is available.

C. Evaluation

To evaluate our system, we set up a small network of four clients (acting as their own puzzle solvers) and a single server protected by a puzzle firewall connected on a single VLAN via a Cisco Catalyst 4006 Gigabit switch. Each client, firewall, and server were dual 1.8GHz Intel Xeon processors with Gigabit Ethernet interfaces.

As discussed in Section IV the expected number of hashes to solve a puzzle is $\frac{\text{difficulty}}{2}$. The number of hashes to generate a puzzle is a constant 2 hashes (1 to hash the answer and 1 to create the issuer cookie), and the number of hashes to verify a puzzle is a constant 1 hash (to match

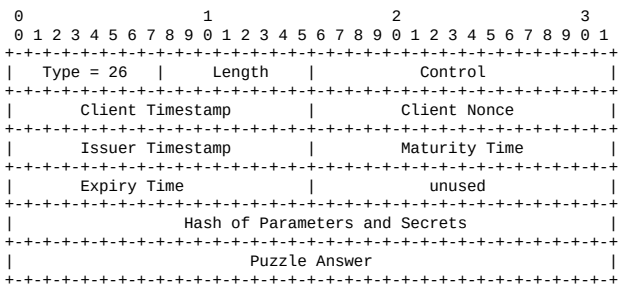
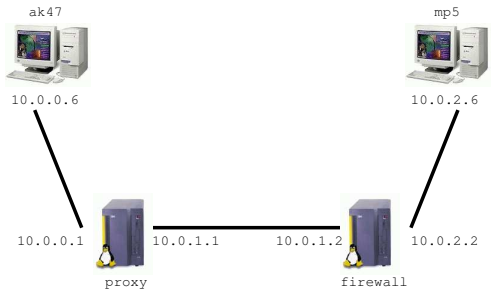


Fig. 8. Answer IP option



Firewall

```
firewall% insmod puzzlenet_mgr.o
firewall% insmod ipt_puzServer.o
firewall% iptables -t mangle -A INPUT -j puzServer
firewall% iptables -t mangle -A FORWARD -j puzServer
```

Proxy

```
proxy% insmod puzzlenet_mgr.o
proxy% insmod ipt_puzClient.o
proxy% iptables -t mangle -A INPUT -p icmp -icmp-type 38 -j puzClient
proxy% iptables -t mangle -A FORWARD -p icmp -icmp-type 38 -j puzClient
proxy% iptables -t mangle -A POSTROUTING -j puzClient
```

Client

```
ak47% telnet mp5 2601
Trying 10.0.2.6...
telnet: Unable to connect to remote host: Connection refused
ak47% telnet 10.0.2.123
Trying 10.0.2.123...
```

`tcpdump` trace

```
17:12:53.632512 10.0.0.6.14698 > 10.0.2.6.2601: S
17:12:53.632566 10.0.1.2 > 10.0.0.6: icmp: type-#38
17:12:56.630212 10.0.0.6.14698 > 10.0.2.6.2601: S
17:12:56.630287 10.0.2.6.2601 > 10.0.0.6.14698: R
17:13:05.455725 10.0.0.6.14699 > 10.0.2.123.23: S
17:13:05.456542 10.0.1.2 > 10.0.0.6: icmp: type-#38
17:13:08.454862 10.0.0.6.14699 > 10.0.2.123.23: S
17:13:14.453935 10.0.0.6.14699 > 10.0.2.123.23: S
```

Fig. 9. Proxy-firewall example

the echoed issuer cookie). To reasonably expect a client to be doing at least as much work as the issuer, the issuer should not create puzzles of difficulty less than 6.

Throttling effectiveness can be measured by the work ratio between the puzzle solver and the puzzle issuer. This can be expressed as $\frac{\text{solution time}}{\text{verification time} + \text{generation time}}$. Using a 32-bit unsigned $\text{difficulty} \geq 6$, the minimum ratio is 1 while the maximum ratio is $\frac{2^{31}}{3}$. Since a hash on the evaluation system takes $1.09\mu s$, we expect our firewall to verify a bad answer and generate a new puzzle in around $3 * 1.09\mu s = 3.27\mu s$. Similarly the maximum difficulty puzzle would be expected to take $1.09\mu s \times 2^{31} = 39.01min$ to solve.

To measure the rate at which a server can verify and

generate puzzles, the clients were configured to flood the server with 64-byte UDP packets with invalid answers as fast as they could. The firewall verified that the answers were invalid and generated a new puzzle for each invalid answer. The firewall's peak sustained throughput over a one minute interval was 182,000 packets per second (or $5.49\mu s$ to verify and generate). This throughput ($\frac{182000 packets}{s} \frac{64B}{packet} \frac{8b}{B} \frac{1Gb}{1073741824b} = 0.087 Gbps$) is slightly lower than expected since there is an unavoidable (yet relatively small) amount of OS contention for the CPU. This shows that the throughput of this software implementation is unsuitable for in-network deployment of puzzle firewalls for all but home networks. However, we are currently investigating a hardware based implementation on the IXP2850 which has special hardware hash units. This device is promising since a hash takes $0.094\mu s$ and we anticipate being able to verify and generate puzzles at Gigabit speeds.

To demonstrate the ability to differentiate between malicious and legitimate clients, we ran another experiment using the same network configuration, but made one of the clients non-responsive by having it refuse to answer any puzzles. A simple controller was implemented to control the amount of traffic accepted by the firewall. The controller targeted a rate of 150,000 packets per second. If the number of packets accepted exceeded or fell underneath the target, the controller scaled the difficulty based on the percentage difference. Figure 10 shows the result of the experiment. After a minute of idling ($t = 60sec$), the non-responsive client floods the server with a packet stream at a rate of around 130,000 packets/sec. As Figure 10(a) shows, since this is below the target forwarding rate, the firewall accepts the packets and does not issue puzzles. After another minute ($t = 120sec$), the three "good" clients begin flooding the server, thus driving the packet rate well beyond 200,000 packets/sec. The firewall quickly enables puzzles and completely wipes out the non-responsive client. While the non-responsive client is still transmitting packets, none of its packets are forwarded by the firewall. As the figure shows, after a brief oscillation, the aggregate throughput of accepted packets for the other three clients remains close to the target rate. Figure 10(b) shows the puzzle difficulty setting at the firewall throughout the experiment. As the figure shows, the difficulty remains at 0 (i.e. no puzzles) while the rate of accepted packets is below the target. As the packet rate increases beyond the target, the difficulty adapts in order to force the packet rate back to the targeted level.

A large part of containing Internet worms is slowing their propogation. Many worms use adaptive port scanning to find new hosts to infect; so by slowing port scans we can slow the propogation of worms. The deployment scenario in the previous section indicates that it is possible to use network puzzles to effectively throttle a port scan. To evaluate this, we compare the time it takes an efficient port scanning tool to scan a server not protected by puzzles to the time it takes the tool to scan a server protected by puzzles of various difficulties. The port scanning tool used was `scanrand`, which can scan an entire class B network in under 4 seconds [31]. Figure 11 shows the results of this experiment; that a ten-fold increase in puzzle difficulty results in a ten-fold increase in scanning time. Without using network puzzles, a scan of 1000 ports took 39ms. At difficulty 100,000 the scan took more than 3 minutes. Extrapolating, puzzles of the maximum difficulty (2^{32}) would force the port scan to take over a month.

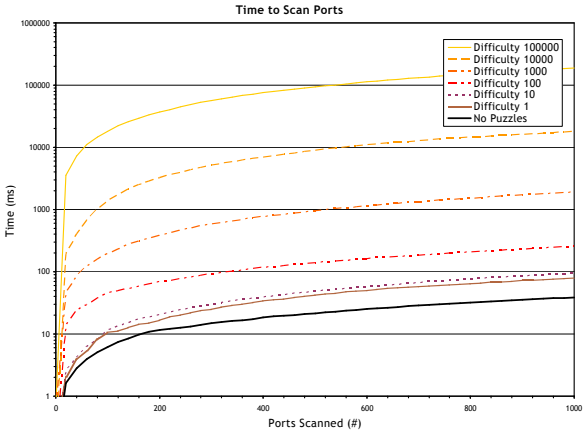


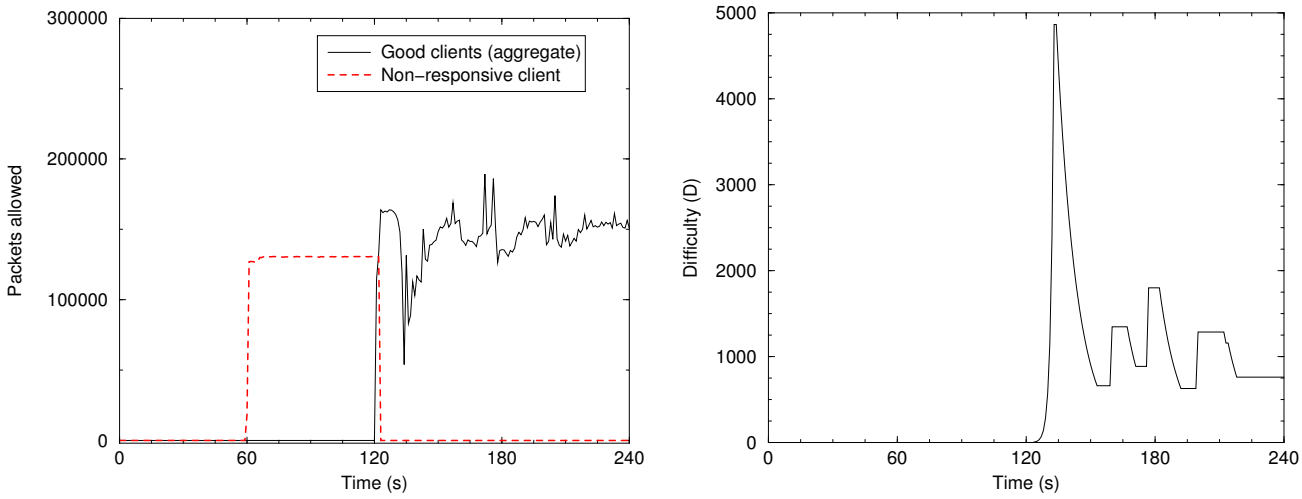
Fig. 11. Ports scanned over time

VI. Discussion

A. Related Work

There have been a large number of efforts related to controlling malicious traffic such as denial of service attacks. One set of approaches focuses on tracing floods back to their sources via targeted packet injection and intelligent packet marking [32], [33], [34], [35], [36], [37]. Another class of approaches is to use pro-active, distributed filtering of packets via direct and indirect methods [38], [39], [40], [41], [42], [43]. These approaches are complementary and can be used in conjunction with puzzle-based approaches.

Cryptographic puzzles themselves were first proposed by Merkle in public key protocols [5]. Since then, puzzles have been applied in specific applications such as authentication protocols [6], [9], [19], e-mail protocols [4], [44], and transport layer protocols [7], [8], [14]. Network layer puzzles do not preclude the use of higher-layer puzzle pro-



(a) Packets accepted

(b) Puzzle difficulty

Fig. 10. Controlling a non-responsive client

tocols. The IP puzzle semantic of "solve a puzzle before I forward your packet" provides additional protection on top of alternative client puzzle protocols where end-host intervention is required. More importantly, however, is the fact that the implementation effectively runs at high-speed, augmenting approaches that rely on some form of network layer protection to guarantee client access [45].

B. Limitations

There are a few known limitations with the current approach that we are working to address. These include:

- *IP header limitations*: The current design and the 40-byte maximum IP header length allows for only a single puzzle answer to be attached on the forward path. While the IPv6 header allows for any number of headers to be used for this purpose [46], we are currently examining IPv4-based mechanisms for supporting multiple puzzle answers per packet in case there are multiple puzzle issuers on an end-to-end path.
- *Eavesdropping attacks*: The lack of a true authentication mechanism means that an eavesdropper along the network path can spoof a puzzle back to the client. For example, on a wireless network, an eavesdropper can capture packets passively, capture the client nonces, and send puzzles back to the victim. While link-layer authentication and encryption can help, this vulnerability should be carefully considered before deployment.
- *Reflector attacks*: Since puzzles consume a non-zero amount of bandwidth, they can be used as part of a reflector attack [18]. Adversaries could spoof a particular source IP address and flood the victim with bogus puzzles. Due to the compact size of the puzzle and the ability to keep such attacks out in the network, however, we argue that IP puzzles do not significantly raise the risk of such attacks compared to spoofed TCP SYN floods.
- *Congestion control*: Puzzles can be used to implement mandatory congestion control. For this to happen, a more sophisticated controller must be designed that can perform robustly in a range of environments. While such controllers exist in the "voluntary" domain of TCP congestion control and active queue management [47], [48], there are no such equivalents in the puzzle domain yet.
- *High-speed router implementation*: Since the protocol and system have been designed with high-speed routers in mind, we are currently implementing a version of it on the fast path of Intel's IXP 2850 network processor [49].
- *Targeted difficulty levels*: The current implementation uses a single, adaptive difficulty level for all of the clients it services. It has been shown that such an approach has many disadvantages including a clear adverse impact on legitimate clients [45]. We are augmenting our system using efficient, high-speed mechanisms [50], [51], [52], [53] for delivering differential puzzles whose difficulties vary based on end-to-end, application-driven information [54], [55], [56]. This work is described in the next sub-section.

C. Future Work: Reputation-Based Networking

The goal of reputation-based networking is to quickly identify malicious clients and place an extremely large computational punishment on all of their communication using network layer puzzles. There is a wealth of locally observable behavior information that can be used to adaptively deliver harder puzzles to clients exhibiting suspicious behavior. For example, intrusion detection systems (IDS) such as Snort [54] as well as application log files can clearly identify systems that are being used for unde-

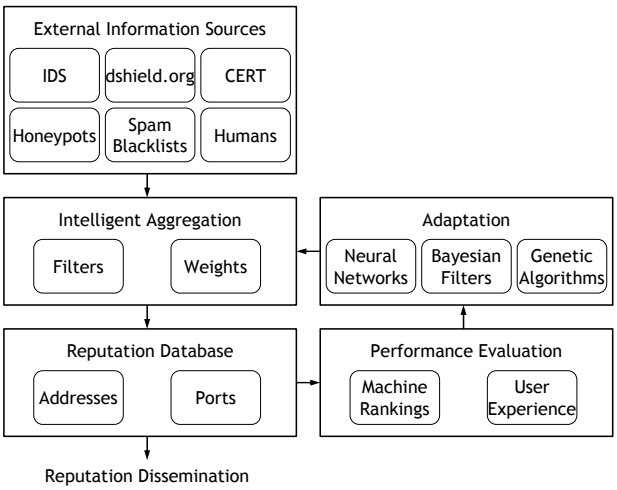


Fig. 12. Managing reputations for network puzzles

sirable purposes. In addition, there is an immense amount of external information that can be used. For example, the DShield service [56] exports a database of information on which ports are being attacked actively and which machines are currently being used to launch attacks.

To perform puzzle difficulty management more intelligently, we are currently building a *Puzzle Manager*: an intelligent agent that aggregates input from a number of information sources in order to determine the reputation of clients and the difficulty of puzzles they must solve to obtain service. Such reputations are then fed into the mechanisms used for punishing malicious clients. We envision that such mechanisms can be used as a form of emergency response to the onset of large-scale cyber-attacks. Specifically, clients with low global reputations will be forced to solve more difficult network puzzles before their packets are routed. Figure 12 outlines the architecture of the reputation-based system we are constructing. As the figure shows, in order to keep up with the changing Internet landscape, the performance of the system must be continuously evaluated against system utilization measurements, machine threat rankings, and user experience reports. Adaptation algorithms will be employed in order to use the feedback to properly adjust the aggregation functions to maximize the system performance. In particular, the system must continuously learn the reliability of individual information sources and adjust the filtering and weighting of information accordingly.

The more interesting research issues focus on the survivability of the system; intelligently thwarting the attempts of malicious clients trying to avoid the punishment mechanisms or subverting the sources of information to render the system completely inaccurate.

VII. Conclusion

Network puzzles are an elegant mechanism for mitigating the effects of undesirable network communication. This paper has described the design and implementation of a network layer puzzle protocol and algorithm that can be used to effectively slow down flooding attacks and port scanning activity. The system allows for high-speed implementations in the fast path of modern network devices, can be flexibly deployed, and is resistant against replay and spoofing attacks.

VIII. Acknowledgments

We would like to thank Tim Sheard for his initial suggestion of sending hints with puzzles, as well as Mark Baugher and Fred Baker for their helpful discussions regarding the protocol and implementation. We would like to thank Raj Yavatkar for the generous support which made this work possible.

References

[1] D. Moore, C. Shannon, and J. Brown, "Code-Red: A Case Study on the Spread and Victims of an Internet Worm," in *Internet Measurement Workshop*, November 2002.

[2] S. Staniford, V. Paxson, and N. Weaver, "How to 0wn the Internet in Your Spare Time," in *11th USENIX Security Symposium (Security '02)*, 2002.

[3] CERT, "CERT Advisory CA-2004-02 Email-borne Viruses," `http://www.cert.org/advisories/CA-2004-02.html`, 2004.

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

[5] R. Merkle, "Secure Communications Over Insecure Channels," *Communications of the ACM*, vol. 21, no. 4, April 1978.

[6] L. von Ahn, M. Blum, N. Hopper, and J. Langford, "CAPTCHA: Using Hard AI Problems for Security," in *Eurocrypt 2003.*, 2003.

[7] A. Juels and J. Brainard, "Client Puzzles: A Cryptographic Defense Against Connection Depletion," in *NDSS*, 1999, pp. 151–165.

[8] D. Dean and A. Stubblefield, "Using Client Puzzles to Protect TLS," in *10th Annual USENIX Security Symposium*, 2001.

[9] T. Aura, P. Nikander, and J. Leiwo, "DOS-Resistant Authentication with Client Puzzles," *Lecture Notes in Computer Science*, vol. 2133, 2001.

[10] J. Leiwo, T. Aura, and P. Nikander, "Towards Network Denial of Service Resistant Protocols," in *SEC*, 2000, pp. 301–310.

[11] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. Wallach, "Security for Peer-to-Peer Routing Overlays," in *Proceedings of OSDI*, December 2002.

[12] M. Abadi, M. Burrows, M. Manasse, and T. Wobber, "Moderately Hard, Memory-bound Functions," 2003.

[13] 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, pp. 46+, 2001.

[14] X. Wang and M. Reiter, "Defending Against Denial-of-Service

Attacks with Puzzle Auctions," in *IEEE Symposium on Security and Privacy*, 2003.

[15] W. Feng, "The Case for TCP/IP Puzzles," in *ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA-03)*, Karlsruhe, Germany, August 2003.

[16] J. Saltzer, D. Reed, and D. Clark, "End-To-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, November 1984.

[17] S. Crosby and D. Wallach, "Denial of Service via Algorithmic Complexity Attacks," in *USENIX Security Symposium*, August 2003.

[18] V. Paxson, "An Analysis of Using Reflectors for Distributed Denial-of-Service Attacks," *Computer Communication Review*, vol. 31, no. 3, July 2001.

[19] W. Aiello, S. Bellovin, M. Blaze, J. Ioannidis, O. Reingold, R. Canetti, and A. Keromytis, "Efficient, DoS-resistant, Secure Key Exchange for Internet Protocols," in *Conference on Computer and Communications Security*, 2002.

[20] D. Bernstein, "SYN Cookies," `http://cr.yp.to/syncookies.html`, 2003.

[21] B. Warner, "EGD: The Entropy Gathering Daemon," `http://egd.sourceforge.net`, 2002.

[22] W. Aiello, S. Rajagopalan, and R. Venkatesan, "Design of Practical and Provably Good Random Number Generators," in *ACM-SIAM Symposium on Discrete Algorithms*, January 1995.

[23] D. Wagner, "Randomness for Crypto," `http://www.cs.berkeley.edu/~daw/rnd/`, 2003.

[24] Intel, "Intel Random Number Generator (RNG)," `http://developer.intel.com/design/security/rng/rngppr.htm`, 2003.

[25] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb, "Building a Robust Network-Processor-Based Router," in *Proceedings of ACM SOSP*, October 2001.

[26] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," *RFC 2246*, January 1999.

[27] K. Fu, E. Sit, K. Smith, and N. Feamster, "Dos and Don'ts of Client Authentication on the Web," in *USENIX Security Symposium*, August 2001.

[28] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock Puzzles and Timed-release Crypto," MIT/LCS/TR-684, 1996.

[29] netfilter/iptables developers, "netfilter/iptables Project," `http://www.netfilter.org`.

[30] J. Postel, "Internet Control Message Protocol," *RFC 792*, September 1981.

[31] D. Kaminsky, "Doxpara: Paketto Keiretsu (scanrand)," `http://www.doxpara.com/read.php/code/paketto.html`, 2002.

[32] H. Burch and W. Cheswick, "Tracing Anonymous Packets to Their Approximate Source," in *USENIX LISA*, December 2000.

[33] S. Savage, D. Wetherall, A. Karlin, and T. Anderson, "Practical Network Support for IP Traceback," in *SIGCOMM*, 2000, pp. 295–306.

[34] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountio, S. Kent, and W. Strayer, "Hash-based IP Traceback," in *SIGCOMM*, August 2001.

[35] D. Song and A. Perrig, "Advanced and Authenticated Marking Schemes for IP Traceback," in *INFOCOM 2001*, 2001, pp. 878–886.

[36] ICMP Traceback Working Group, "ICMP Traceback (itrace)," `http://www.ietf.org/html.charters/itrace-charter.html`, 2002.

[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*, May 2003.

[38] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling High Bandwidth Aggregates in the Network," *Computer Communication Review*, vol. 32, no. 3, July 2002.

[39] H. Jamjoom and K. Shin, "Persistent Dropping: An Efficient Control of Traffic Aggregates," in *SIGCOMM*, August 2003.

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

[41] D. Andersen, "Mayday: Distributed Filtering for Internet Services," in *USITS*, March 2003.

[42] K. Lakshminarayanan, D. Adkins, A. Perrig, and I. Stoica, "Taming IP Packet Flooding Attacks," in *Hot Topics in Networks (HotNets-II)*, 2003.

[43] S. Ioannidis, A. Keromytis, S. Bellovin, and J. Smith, "Implementing a Distributed Firewall," in *ACM Conference on Computer and Communications Security*, 2000, pp. 190–199.

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

[45] V. Gligor, "Guaranteeing Access in Spite of Service-Flooding Attacks," in *Security Protocols Workshop*, April 2003.

[46] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," *RFC 2460*, December 1998.

[47] Van Jacobson, "Congestion Avoidance and Control," in *Proceedings of ACM SIGCOMM*, August 1988, pp. 314–329.

[48] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *ACM/IEEE Transactions on Networking*, vol. 1, no. 4, pp. 397–413, August 1993.

[49] Intel, "Intel IXP2850 Network Processor," `http://www.intel.com/design/network/products/npfamily/ixp2850.htm`, 2003.

[50] D. Lin and R. Morris, "Dynamics of Random Early Detection," in *Proceedings of ACM SIGCOMM*, September 1997.

[51] W. Feng, D. Kandlur, D. Saha, and K. Shin, "Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness," in *Proc. of INFOCOM*, April 2001.

[52] P. McKenney, "Stochastic Fairness Queueing," in *Proceedings of IEEE INFOCOM*, March 1990.

[53] J.L. Rexford, A.G. Greenberg, and F.G. Bonomi, "Hardware-Efficient Fair Queueing Architecture for High-Speed Networks," in *Proceedings of INFOCOM*, March 1996.

[54] M. Roesch, "Snort - Lightweight Intrusion Detection for Networks," in *Proceedings of the 13th Systems Administration Conference (LISA '99)*, 1999.

[55] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," in *10th Annual USENIX Security Symposium*, January 1998.

[56] DShield.org, "Distributed Intrusion Detection System," `http://www.dshield.org`, 2002.