

# Deamplification of DoS Attacks via Puzzles

Jacob Beal and Tim Shepard  
 jakebeal@mit.edu, shep@alum.mit.edu

**DRAFT: Please do not redistribute**

October 15, 2004

*Abstract*—Puzzles have been proposed as a mechanism to deamplify denial of service attacks against a server’s memory and processing resources. For example, HIP implements a cookie puzzle mechanism to protect the server from wasting resources performing Diffie-Hellman exponentiation in response to spurious requests. We examine cookie puzzle mechanisms of this type.

We find that careful attention is needed in server implementation to ensure that an attacker does not retain opportunities to amplify the attack despite the puzzle mechanism, and present a design which addresses these issues. We compare vulnerability to bandwidth and processing attacks, determining when one dominates the other. Finally, we quantify the deamplification of DoS attacks provided by a cookie puzzle mechanism and determine the best setting for puzzle difficulty under a steady-state attack.

## I. INTRODUCTION

Denial of service attacks can be targeted at any exhaustible resource, such as bandwidth<sup>1</sup>, memory, or processing power. Although recently most attention has been focused on denial of service attacks which target bandwidth, the potential impact of attacks on processing power is as great or greater in some cases, such as key exchange protocols (e.g. Diffie-Hellman key agreement protocol).

Key exchange protocols used to establish session keys using public key cryptography require the responding computer to perform an expensive exponentiation operation. Because an expensive operation must be performed before the responder can authenticate the identity of the initiator, an attacker might consume the responder’s processing power with a flood of bogus requests.

This is an instance of the more general problem of protecting processing power, which is simpler in this case because every request consumes an approximately equal amount of computation. Additionally, a solution in the case of key-exchange protocols may open up a much wider range of potential solutions to the general

<sup>1</sup>We use the term bandwidth throughout this paper not in its literal sense, but rather in its figurative sense, meaning communication capacity, that has become commonplace in writing on computer networking.

problem, because authenticated connections might be used to establish accountability for resources consumed.

One defense, used by IKEv2[8], is for the responder to perform a return routability test before committing resources to the initiator’s request. Although this prevents simple spoofing, return routability cannot distinguish between a single attacking machine which controls multiple IP addresses (easily available in IPv6) and a group of legitimate users, nor between a single attacking machine and a large pool of legitimate users sharing a single address via a NAT<sup>2</sup> box (common in IPv4).

Thus even with a return routability test, an attacker willing to expose one or more routable locations (e.g. hijacked machines) can amplify an attack by simulating a large group of legitimate users, thereby tricking the responder into a disproportionate expenditure of resources.

The HIP protocol[12] attempts to solve this problem by means of a cookie puzzle mechanism[3], in which the cookie sent by the responder to test return routability includes a cryptographic puzzle which the initiator must solve in order to obtain service. The hope is that, by adjusting the difficulty of the puzzle, the responder can deamplify attacks, while not significantly burdening legitimate clients.

In this paper we address three issues of cookie puzzle system design and implementation:

- **Design:** We find that the mechanisms suggested in the HIP drafts are, by themselves, insufficient. Careful server implementation is required to ensure that an attacker does not retain opportunities to amplify the attack despite the puzzle mechanism. In Section III we explain the threats and present a server design which uses persistent-dropping to defend itself.
- **Dominance:** The more asymmetric an attack, the more attackers can gather sufficient resources, and the less exposed an attacker must be in order to execute it. In Section V we compare the asymmetry of attacks against processing resources and band-

<sup>2</sup>Network Address Translation

width. When processing vulnerability dominates, cookie puzzle mechanisms are useful in decreasing the asymmetry of attack.

- **Deamplification:** In Section VI, we quantify the reduction of attack intensity provided by a well-designed cookie puzzle system. Furthermore, we determine how to set puzzle difficulty to optimize service to legitimate clients under steady-state attack conditions.

## II. THE HOST IDENTITY PROTOCOL

The Host Identity Protocol Architecture [13] is proposed within the IETF to fill a gap between DNS names and IP addresses. It provides for a new namespace, Host Identity, to enable trusted communication between machines which do not have stable network addresses. One of its components is the Host Identity Protocol (HIP) [12], which allows two hosts to establish authenticated communication by means of a four-packet exchange.

At the present time, there are four open source implementations that have demonstrated interoperability [1], [14], [9], [10]. To date, interoperability demonstration has been the goal of these implementation efforts, and these implementations are not yet complete with respect to HIP's designed defenses against DoS attacks.

We will sketch the key points of a HIP exchange for this paper; for a proper treatment, see [12].

First, the client sends a request to the server, asking to establish an association. In response, the server initiates a Diffie-Hellman key exchange, conducted in the second and third packets. Rather than risk its memory by keeping any state<sup>3</sup>, the server includes all necessary association information in its reply packet. The client sends this information back to the server with its half of the key exchange, and the server completes the association.

To protect itself from fake packets and colluding attackers, the server includes a cookie consisting of a random number and a correlated value, partially determined by the client's identity and the time of the request.<sup>4</sup> This allows the server to verify that the returned cookie is genuine and was sent from it to this particular client before running the computationally expensive key exchange calculation following the third packet or allocating memory for the association.

The optional puzzle component of the protocol is implemented by requiring the client to find a third

<sup>3</sup>Keeping state risks attacks like the SYN attack, in which the attacker continually opens and drops TCP connections, overflowing the victim's memory with stale connections.[4]

<sup>4</sup>The HIP specification suggests a table of precomputed packets, selected by hashing the client's identity.

number, such that a cryptographic hash of the packet, with this third number included, has its last  $k$  bits all zero. This forces the client to "pay" by brute-force searching through an expected  $2^k$  numbers to find one which produces enough zeros. The cost to the server is trivial, and the cost to the client can be adjusted from nothing to prohibitive by setting the value of  $k$  in the range from 0 to 64 bits.

## III. COOKIE PUZZLE SYSTEMS

Cookie puzzles can generally be used to protect a server's memory and processing power from DoS attacks. A general cookie puzzle system consists of two components: a four-packet protocol (after the model of HIP) backed by a complementary server implementation which guards against the opportunities for attack amplification presented by the protocol.

We will hereafter refer to the action enabled by the protocol as the transaction (e.g. Diffie-Hellman key agreement in HIP).

### A. Cookie Puzzle System Model

The purpose of a cookie puzzle is to defend server resources. We expect any well-designed cookie puzzle system to exhibit the following properties with respect to server resources (based on our interpretation of the design goals of HIP's puzzle mechanism).

*a) The client expends resources on the transaction before the server.:* The server's cost in the transaction is assumed to be vastly dominated by the post-puzzle computation (e.g. the Diffie-Hellman exponentiation in HIP). In other words, sending the puzzle cookie and checking its solution must be nearly free in amortized cost to the server, and the server checks that the solution to the puzzle is correct before performing its costly part of the transaction.

*b) Every transaction request has an equal chance of being serviced.:* Careful design of the server is required to prevent an attacker from obtaining preferential service for its transactions (e.g. by resending the same transaction request aggressively).

*c) Every transaction processed by the server must be associated with at least one unique valid puzzle solution.:* The cryptographic hash allows a carefully designed server to reject fake packets, modification of the cookie, or reuse of the solution by the same attacker in a replay attack, or by conspiring attackers in a cookie-jar attack.<sup>5</sup> Good protocol design and server security

<sup>5</sup>The "cookie-jar" strategy has attackers sharing information to amplify an attack. In this case, one attacker would solve the puzzle and then many attackers would use its single solution to transact with the server.

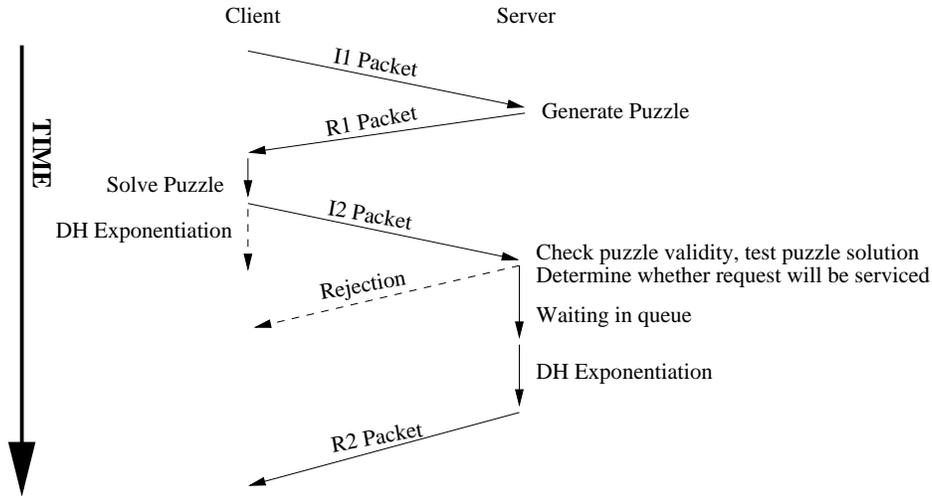


Fig. 1. Host Identity Protocol Key Exchange. The client starts the exchange by asking for a connection. The server responds by starting the key exchange, accompanied by a cookie carrying all state for the exchange, and a cryptographic puzzle. Once the client has solved the puzzle, it sends the answer along with the cookie and the other half of the key exchange. The server checks that everything is in order, then does its expensive computation for the key exchange and establishes the connection. Note that all state for the exchange is stored in the packets, protecting the server’s memory, and that the server waits until it has a valid response from the client before doing its expensive computation.

are assumed to prevent other avenues of circumventing the puzzle requirement. We will assume this ratio is exactly one, maximizing the potential of the attacker and simplifying our analysis.

d) *No server memory is consumed by a transaction until after the client has delivered a valid puzzle solution.*: The cookie contains all of the state needed for the server to continue to process the transaction, so memory-consuming attacks against the server (e.g. the SYN attack) are not possible without engaging in puzzle solving.

### B. Server Implementation

In order to effectively use a cookie puzzle protocol to prevent an attacker from amplifying an attack, careful server design is required. The key requirement is that the server must be **fair** in servicing transaction requests: every unit of puzzle-solving work must have an equal chance of earning service for that transaction, and no puzzle solution may be serviced more than once.

There are four key strategies by which an attacker can attempt to induce unfairness:

- **Counterfeiting**: The attackers may send invalid cookies or puzzle solutions.
- **Time Shifting**: An attacker might prepare for an attack by solving puzzles and caching the results, then expend them to amplify its attack.
- **Collusion**: Attackers may share solutions (A “cookie jar” strategy) so that one valid solution can

be used by many attackers.

- **Repetition**: The attacker may send the same valid solution many times.

At the core of the design is a queue containing transaction requests waiting for service. The queue has a fixed size, and any transaction request which attempts to enter when the queue is full is dropped. If entry into the queue is fair, then servicing of transaction requests is fair as well, so our defensive design centers around this issue.

Counterfeiting, time-shifting, and collusion may be dealt with as recommended in the HIP protocol: the server maintains a table of precomputed response packets, which it selects from by hashing the client’s identity. This protects against counterfeit cookies and collusion, because the contents of the client’s transaction request can be checked against the precomputed packet for that identity, and discarded if the cookie fails to match. Similarly, counterfeit puzzle solutions are easily detected because they do not produce enough zeros when hashed.

At regular intervals, a new table of response packets is generated, to protect against time-shifting attacks. Old tables are retained for a few intervals, to allow for late-arriving client responses.

Finally, we must defend against repetition. By repeated transmission of the same valid transaction request, an attack can attempt to fill the queue with multiple copies of the same transaction request. If only one copy of a request is allowed in the queue, then repetition

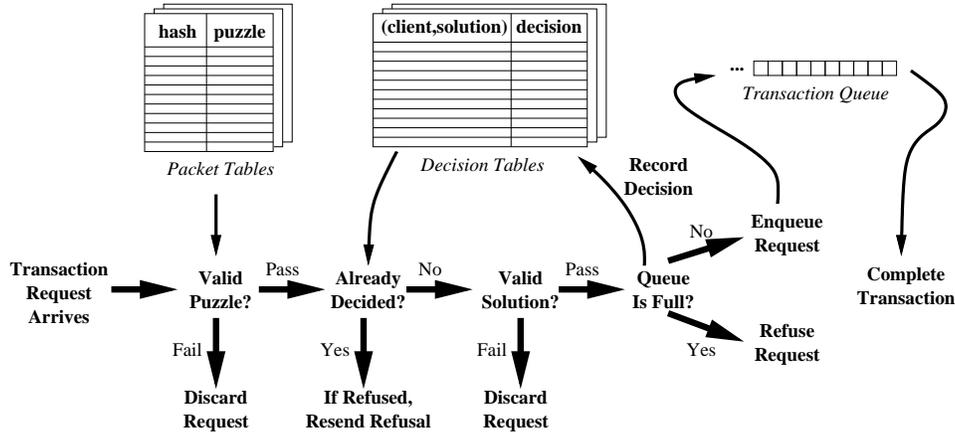


Fig. 2. Server implementation. A transaction request arriving at the server goes through several tests before being enqueued for service. When a request passes all tests and attempts to enter the queue, the server records whether it succeeds, so that repetitions of the request cannot attempt to enter the queue again. Old decision records are discarded when the associated set of response packets expires.

can still create unfairness by raising the chance that the repeated request will be the first to arrive and seize a newly free slot in the queue.

To defend against repetition, we use a mechanism similar to the persistent dropping system in [7], in which our server records every decision it makes. For each table of packets, there is a table of decisions, which associates the client identity and solution with whether the transaction request is dropped or enqueued. The decision tables can be discarded when the associated table of response packets is discarded, since those packets cannot receive service anymore. When a transaction request arrives for which the decision is already in the table, the server need not consider it any further.

Using dynamic perfect hashing[6], the size of the table is linear in the size of the key/value pairs stored therein, and insertion and lookup require constant amortized time. Such a table may be quite reasonable in size, since each entry requires only 192 bits for the key (128 bits identity, 64 bits puzzle solution) and one bit of value to record the decision: at a rate of 1 million transaction requests per second, with puzzles valid for one minute, a server needs approximately 2 GB of memory. This could be decreased by two orders of magnitude, at the cost of occasional false positives, by means of a Bloom filter.

We combine these mechanisms as shown in Figure 2, such that each successive stage requires more imitation of a valid transaction request on the part of the attacker. As an additional service to the client, if bandwidth is available, the server sends notification of dropped transaction requests so that legitimate clients may distinguish

between poor network conditions and server overload.<sup>6</sup>

#### IV. ATTACK MODEL

There are three players in our attack scenario: attackers, legitimate clients, and the server. In this scenario, the server is providing content or services to a large client base where each individual client transacts with the server infrequently (for example, the server might be a public key server, searchable database, or Internet banking system).

The attacker attempts to disrupt service to the legitimate clients by flooding the server with apparently legitimate service requests. By assuming multiple identities per machine, the attacker can match its apparent transaction rate per identity to that of legitimate clients, further camouflaging its attacks and rendering the attack undetectable except as an unexpected surge of service demand.

Network resources are assumed to be large enough to handle all traffic: the resource under attack is server computation. To resist this attack, the server is only allowed to adjust puzzle difficulty.

We model this as follows:

##### A. Legitimate Client Pool:

There are an infinite number of identical legitimate clients, each of which performs only one transaction. Legitimate clients originate transactions at a fixed rate of  $C$  transactions per second. Each transaction requires  $t$  instructions of computation by the client, so if the

<sup>6</sup>This functionality has been added as an option in an upcoming draft of the HIP protocol.

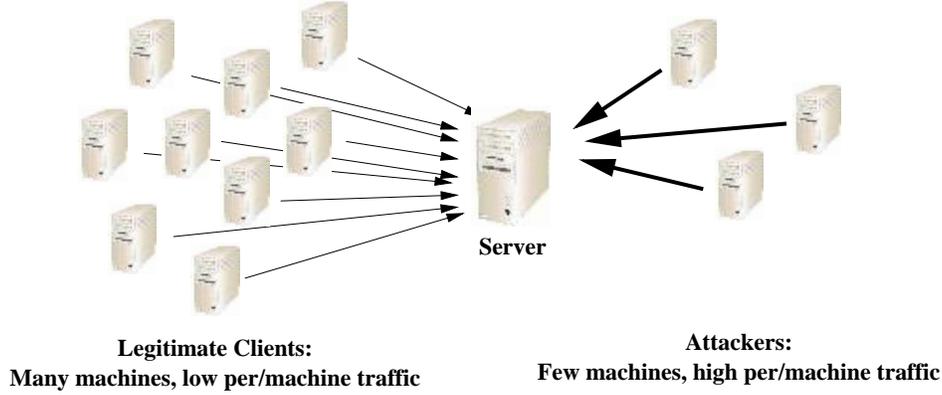


Fig. 3. An unfilterable DDoS attack on a server with a large low rate client base. In this scenario, the attacker floods the server with apparently legitimate service requests, in which each attacking machine pretends to be many legitimate clients. The attack traffic cannot be distinguished from the legitimate traffic because each attacking *identity* only generates a small amount of traffic.

client can process  $f$  instructions per second, then each transaction costs the client  $t/f$  seconds of computation. In addition, the server supplies a puzzle expected to take  $P$  instructions to solve, for a total of  $\frac{P+t}{f}$  seconds of client computation per transaction.

Client bandwidth is not a constraint because traffic per client is very small.

#### B. Attacker Pool:

The attacker pool consists of a fixed number  $N$  of zombie machines<sup>7</sup> identical to legitimate client machines.<sup>8</sup> The attackers freely originate bogus transactions capable of tricking the server into fully expending resources before determining that the transaction is bogus.

Since the attackers only need to solve the puzzle in order to trick the server into expending resources, each bogus transaction costs the attacker only the  $P/f$  seconds of computation expended in puzzle-solving. The attacker pool generates bogus transactions as fast as possible, so zombie transactions arrive at the server at the rate  $Z = Nf/P$  transactions per second. Finally, there is some small base cost for overhead on an attack which we will model as a lower limit on effective puzzle difficulty.

Each zombie is connected to the internet via a link with bandwidth  $b$  bits per second.

#### C. Server:

The server processes  $F$  instructions per second, and each transaction takes  $T$  instructions to process, so the

server takes  $T/F$  seconds of computation to process each transaction. Thus it can sustain a maximum arrival rate of  $F/T$  transactions per second.

The server makes fast, persistent decisions as to whether a transaction request will be serviced, after the design in Section III-B.

The server can attempt to regulate transaction arrival rate by adjusting puzzle difficulty  $P$  over a continuous range.<sup>9</sup>

The server is connected to the internet via a link with bandwidth  $B$  bits per second.

#### D. Transaction Delay:

The total time for a successful transaction is the sum of:

- a network round trip for the first packet exchange
- time spent by the client solving the puzzle ( $P/f$ )
- a network round trip for the second packet exchange
- time spent waiting in the server's queue
- time spent by the server processing the transaction ( $T/F$ )
- time spent by the client completing the transaction ( $t/f$ )

(Processing time for the first packet exchange is assumed to be negligible for both the client and server.)

Let the maximum latency in the queue be  $l_q$ , and the expected latency for a successful network round trip be  $l_n$ .

For purposes of our analysis, we assume the queue is always almost full (otherwise, no transactions are in

<sup>7</sup>compromised machines used as pawns by the attacker

<sup>8</sup>A powerful attacker machine can be represented as many zombie machines. For example, if legitimate clients are PDAs, then a powerful desktop machine might be equivalent to 100 zombie machines.

<sup>9</sup>Although HIP restricts the difficulty to powers of two, finer adjustments are possible by allowing ranges.

Variable	Meaning (units)
$P$	Expected puzzle difficulty (instructions)
$N$	Number of zombies (pure number)
$f$	Client/Zombie clock frequency (instructions/second)
$b$	Client/Zombie bandwidth (bits/second)
$t$	Client transaction difficulty (instructions/transaction)
$F$	Server clock frequency (instructions/second)
$B$	Server bandwidth (bits/second)
$T$	Server transaction difficulty (instructions/transaction)
$C$	Load from clients (transactions/second)
$Z$	Load from zombies (transactions/second)
$L$	Load total (transactions/second)
$A$	Expected attempts per transaction (pure number)
$p$	Probability of a successful transaction (probability)
$q$	Probability of a client becoming discouraged (probability)
$l$	Expected latency from network and queue (seconds)
$l_q$	Maximum latency in the queue (seconds)
$l_n$	Expected latency for a successful network round-trip (seconds)
$d$	Expected delay (seconds)
$d_s$	Expected time per success under heavy load (seconds)
$d_f$	Expected time per failure (seconds)
$d_{max}$	Allowable expected delay (seconds)
$P_{max}$	Maximum single-puzzle difficulty (instructions)

TABLE I  
SUMMARY OF VARIABLES

danger of being dropped). Thus the expected delay for a successful transaction with a full queue is:

$$d_s = \frac{t + P}{f} + \frac{T}{F} + l_q + 2l_n \quad (1)$$

A transaction fails when the server is overloaded and refuses the transaction. Thus the expected delay for a failed transaction is:

$$d_f = \frac{P}{f} + 2l_n \quad (2)$$

The client may retry a failed transaction, but will only tolerate an expected delay  $d_{max}$  before losing patience and abandoning a transaction (consequently puzzle difficulty is limited to  $P_{max}$ .<sup>10</sup>) Thus we will consider the attacker to have disrupted service if the expected delay for legitimate clients is raised above  $d_{max}$ .

The measure of a server's robustness, then, is the minimum size of an attacker's pool of zombie machines required to disrupt service to the client.

<sup>10</sup> $P_{max} = (d_{max} - (\frac{T}{F} + l_q + 2l_n))f - t$

## V. WHEN SHOULD SERVER RESOURCES BE DEFENDED?

A cookie puzzle protocol protects only server resources and not network resources, so if the attacker has sufficient resources to saturate the server's bandwidth, then the cookie puzzle is irrelevant, since transactions cannot reach the server. With a few back of the envelope calculations, however, we can see that for many situations, attacks against processing are more effective than attacks against bandwidth.

Assuming that the attackers are well distributed through the internet and that the server's defenders filter aggressively, ultimately a network attack comes down to a question of whether the group of zombies has more bandwidth than the server. More precisely, the zombies alone can saturate the server's bandwidth when:

$$Nb \geq B \quad (3)$$

By contrast, a group of zombies can saturate the server's processing when:

$$N \frac{f}{t} \geq \frac{F}{T} \quad (4)$$

Combining these two equations, we see that bandwidth attacks dominate when the server/zombie asymmetry is greater for processing than for bandwidth:

$$\frac{B}{b} < \frac{F}{T} / \frac{f}{t} \quad (5)$$

Since the base cost of a bogus transaction is generally extremely low, services with a high server transaction cost may be much more vulnerable to processing attacks than bandwidth attacks. It is this class of situations which motivate the use of puzzles to defend server resources.

#### A. Example

Consider a server connected to the internet via a one gigabit per second link (presumably there are other services sharing the link which justify its capacity), which is capable of doing 1000 Diffie-Hellman exponentiations per second and is under attack by zombies with one megabit per second links. The bandwidth asymmetry is  $B/b = 1000$ , so a pool of approximately 1000 zombies is necessary to saturate the server's bandwidth.

If each transaction requires a Diffie-Hellman exponentiation by the server and it takes only one millisecond of processing to transmit a bogus transaction request, then the asymmetry is very low:  $\frac{F}{T} / \frac{f}{t} = 1000/1000 = 1$ . This means that a single zombie can consume the server's entire processing capacity.

A cookie puzzle protocol could increase the asymmetry by requiring extra processing by the attacker for each transaction. The point of diminishing returns is reached when bandwidth asymmetry and processing asymmetry are equal, at which point it becomes more effective to attack bandwidth than processing.

## VI. DEAMPLIFICATION VIA PUZZLE DIFFICULTY

By increasing puzzle difficulty, the server can reduce the load from attacker generated transactions. If the zombie machines are few enough, the server can minimize delay for legitimate clients while never refusing a legitimate transaction by adjusting puzzle difficulty to deamplify the attacker transaction rate to match its unused processing capacity.

Otherwise, if the attack cannot be deamplified to match unused server capacity (either because the legitimate clients are already overloading the server, or because sufficient deamplification would require longer delays than the clients are willing to tolerate) then some transactions must be refused. In this situation setting puzzle size to the maximum allowable difficulty,  $P_{max}$ , produces the least disruption for legitimate clients, whether or not they are willing to retry transactions which have been refused.

#### A. Forcing a Sustainable Arrival Rate

The simplest way for a server to deal with an attack against CPU resources is to raise the puzzle cost such that transactions arrive at sustainable rates.

The server is fully loaded when the arrival rate of legitimate and attack transactions equals its processing power

$$(C + Z) = F/T. \quad (6)$$

Substituting  $Z = Nf/P$  and solving for puzzle difficulty, we find the unsurprising result

$$P = \frac{Nf}{F/T - C}. \quad (7)$$

In other words, the puzzle difficulty should be set to fit the attack transactions precisely into the arrival capacity not used by the legitimate traffic. Note that if legitimate traffic is exceeding the server's capacity already ( $C > F/T$ ), then this strategy is not usable.

To disrupt service, an attacker needs to force expected delay above  $d_{max}$  for a single transaction. Setting puzzle difficulty to the maximum one-shot puzzle,  $P_{max}$ , Equation 7 can be rearranged to find the maximum number of zombie machines that can be tolerated,

$$N = \frac{P_{max}}{f}(F/T - C). \quad (8)$$

In other words, if the client will tolerate solving an  $n$ -second puzzle once, the attacker needs  $n$  zombie machines for every transaction per second of server load unoccupied by legitimate transactions. For a server which normally runs well under capacity, this is a massive resiliency.

1) *Example Scenarios:* Consider a database server capable of handling 1000 transactions per second ( $F/T = 0.001$ ), and a 30% load from legitimate clients ( $C = 300$ ). If no puzzle is used (assuming a base  $P/f = 0.001$ ) then a single attacking machine can pump the server to 130% load, causing legitimate transactions to be dropped. If the client will tolerate a half-second puzzle ( $P_{max}/f = 0.5$ ), then an attacker needs 350 zombie machines to affect legitimate transactions. If the client will tolerate a ten second puzzle ( $P_{max}/f = 10$ ), then it is resilient against attacks of up to 7000 zombie machines.

A large server farm has an even more staggering resilience. If a large e-commerce company is running a server farm capable of one million transactions per second ( $F/T = 10^{-6}$ ), then at 30% load ( $C = 300,000$ ), with no puzzle ( $P/f = 0.001$ ) only 700 zombies are needed to affect service. If a two second puzzle

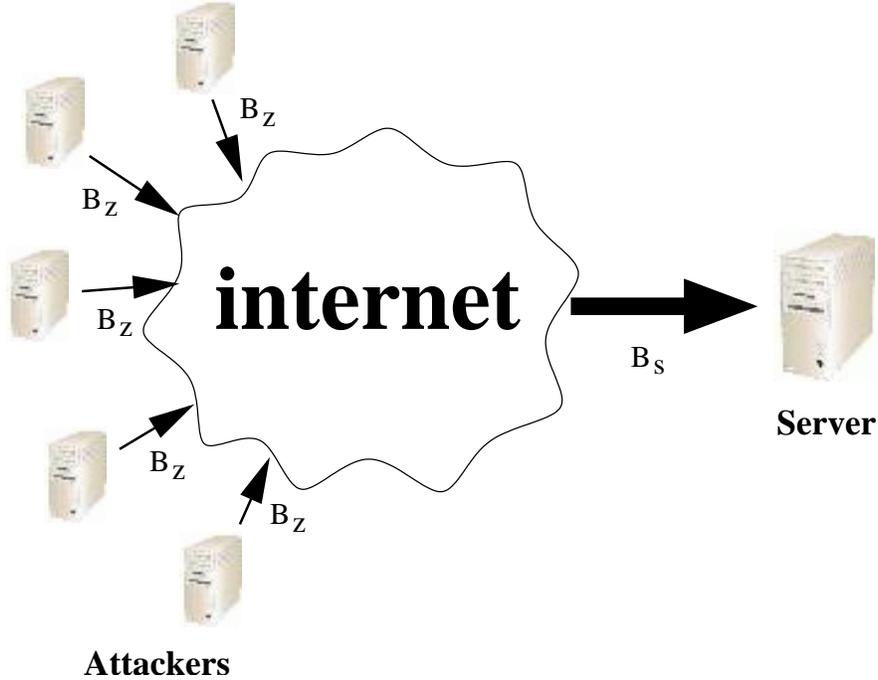


Fig. 4. Bandwidth attack. An attack on the server’s network connection requires the zombies to have more aggregate network capacity than the server. If the server is providing a processing-intensive service, then fewer zombies may be necessary for an attack on processing than for an attack on bandwidth. Cookie puzzle protocols raise the cost of attacking processing, forcing attackers to use more resources or to switch back to bandwidth attacks.

( $P_{max}/f = 2$ ) is used then the attacker needs a pool of 1.4 million zombie machines to affect service.

As the legitimate client load rises, the resilience drops sharply. If the e-commerce company in the previous example is experiencing peak traffic and running at 95% load ( $C = 950,000$ ), then the attacker only needs 100,000 machines to affect service, an order of magnitude less.

2) *Infinitely Persistent Clients*: Legitimate clients whose transactions are refused may try again, so a refused transaction is not necessarily a failure. Thus, decreasing puzzle difficulty might reduce the average delay for a successful transaction because only clients whose transactions are refused would have to solve more than one puzzle.

It turns out, however, that the increased transaction rate caused by clients retrying refused transactions raises the expected time before success enough to outweigh the benefit from decreasing puzzle difficulty.

If the client retries until successful, the server will face repeat traffic from clients which were unsuccessful on their first attempt, raising its load. We can find the steady-state transaction rate  $L$  and expected number of attempts per transaction  $A$  by means of a pair of equations,

$$L = Z + CA \quad (9)$$

and

$$A = \begin{cases} \frac{L}{F/T} & \text{if } L \geq F/T \\ 1 & \text{if } L \leq F/T \end{cases} \quad (10)$$

In essence, the expected number of attempts is equal to the normalized server load for an overloaded server, since transactions requests are discarded randomly. In steady-state, attacker load is constant and the legitimate load is the arrival rate for new transactions times the expected number of attempts per transaction. Solving for the steady-state attempts  $A$  on an overloaded system ( $L \geq F/T$ ), we find

$$A = \frac{Z}{F/T - C}. \quad (11)$$

The equations converge to this value whenever legitimate traffic is less than the capacity of the system. If legitimate traffic overloads the server, then it is never possible for all transactions to be serviced.

How, then, should we set puzzle difficulty  $P$  to minimize delay for the client? By Equation 11, the client is expected to succeed after  $Z/(F/T - C)$  attempts, with each failure costing  $d_f$  seconds and the success costing

$d_s$  seconds. For an overloaded system, then, the expected time to produce a successful transaction is

$$d = d_f(A - 1) + d_s \quad (12)$$

$$d = (P/f + 2l_n) \frac{Z}{F/T - C} + T/F + l_q + t/f. \quad (13)$$

Substituting  $Z = Nf/P$  and distributing yields

$$d = (P/f + 2l_n) \frac{Nf/P}{F/T - C} + T/F + l_q + t/f \quad (14)$$

$$d = \frac{N}{F/T - C} + \frac{2l_n Nf}{P(F/T - C)} + T/F + l_q + t/f. \quad (15)$$

Note that the delay predicted by this equation decreases monotonically as puzzle difficulty increases. Thus, for overloaded systems, increasing puzzle difficulty decreases delay, until the point when the puzzle difficulty is high enough that the system is no longer overloaded (at which point  $L < F/T$  and the above equation is no longer valid). So minimizing client delay for a server which is not overloaded by legitimate traffic alone may be reduced to the case of finding the least puzzle difficulty that yields a sustainable arrival rate (Equation 7).

Thus, if clients retry until they succeed, refusing a transaction is never better than setting  $P$  to force a sustainable arrival rate.

### B. Limiting Service Disruption

If the attacker has enough zombie machines, then forcing a sustainable arrival rate may lead to a puzzle difficulty that exceeds the patience of the clients. Worse yet, if the server is already fully loaded (or worse) by legitimate client transactions, then it may be impossible to set the puzzle difficulty to force a sustainable arrival rate.

In this situation, the goal is instead to minimize the fraction of legitimate clients whose transactions are not served.

We will first examine the behavior of the system for clients that only try once. We then generalize the client with a probability  $q$  of abandoning a transaction each time it is refused. This generalized model is equivalent to one-shot clients when  $q = 1$  and infinitely persistent clients when  $q = 0$ . We will show that for any value of  $q$ , the best result is achieved by setting puzzle difficulty as though clients were one-shot.

1) *One-Shot Clients*: If clients do not retry refused transactions, then any transaction which is refused fails.

Assuming that the puzzle size has been raised to the maximum,  $P_{max}$ , the probability of a successful transaction  $p$  on an overloaded server ( $(C + Z) \geq F/T$ ) is inversely proportional to the server's load,

$$p = \frac{F/T}{C + Z}. \quad (16)$$

Substituting  $Z = Nf/P$  and setting  $P = P_{max}$ , we find

$$p = \frac{F/T}{C + Nf/P_{max}}. \quad (17)$$

Thus, it may take very few zombies to impact service on a server which is already heavily loaded with legitimate transactions. The number of zombies necessary to have a large effect on service, however, is roughly the same no matter the legitimate client load. We illustrate this in Figure 5, which shows the service disruption inflicted by various different size attacker pools on the example servers from Section VI-A.1. Note that the behavior of small and large servers is identical when load is normalized for server capacity.

2) *Discourageable Clients*: In the more general case, clients will retry refused transactions a few times before becoming discouraged and giving up.<sup>11</sup> This complicates the predictions of service disruption because, while there are more chances for a client to succeed in its transaction, the retries add load to the server, making any particular attempt more likely to be refused.

We model *discourageable* clients as having a probability  $q$  of giving up after each unsuccessful transaction attempt.<sup>12</sup> The probability that a given transaction attempt will succeed is the inverse of the normalized load,  $\frac{F/T}{L}$ , so the probability of a client stopping (either by succeeding or by giving up) after a particular transaction attempt is

$$P(stop) = P(succeed) + P(fail) * P(discourage) \quad (18)$$

$$P(stop) = \frac{F/T}{L} + (1 - \frac{F/T}{L})q. \quad (19)$$

and therefore, considering this as a Bernoulli process, the expected number of attempts per transaction  $A$  for discourageable clients is

<sup>11</sup>We can expect this behavior from either a human or a polite automated system.

<sup>12</sup>As previously noted, this is a generalization with special cases  $q = 1$  being one-shot clients and  $q = 0$  being infinitely persistent clients.

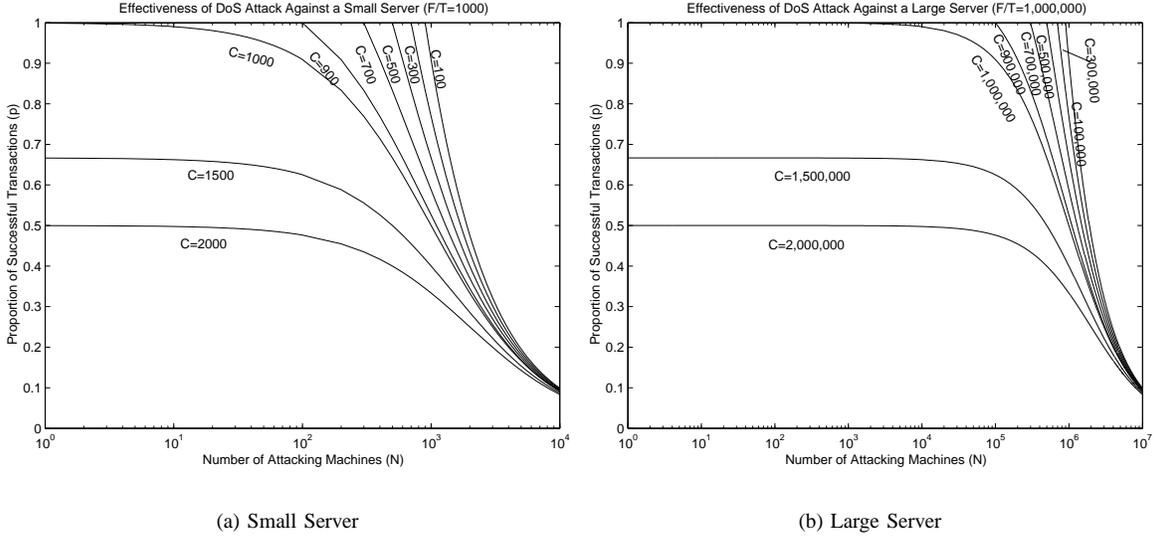


Fig. 5. Disruption of service for an example small and large server under various loads of legitimate traffic ( $C$ ), with a 1-second puzzle ( $P/f = 1$ ) (Calculating probability of service by  $p = \min(1, \frac{F/T}{C + Nf/P})$ , after Equation 17). The attack has no effect on the probability of a legitimate transaction being served until the server's processing capacity is fully saturated. Significant degradation of service, however, requires a similar large number of zombies no matter the server load. Note that the two graphs would be identical if  $C$  and  $N$  were normalized for server capacity.

$$A = \frac{L}{(L - F/T)q + F/T}. \quad (20)$$

Assuming that the maximum expected delay tolerated by the client is  $d_{max}$ , the larger the number of attempts, the less difficult the puzzle that can be solved during each attempt. Since each attempt involves additional delay from network overhead as well, the maximum puzzle difficulty per attempt,  $P'_{max}$ , is strictly less than the maximum one-shot puzzle difficulty divided evenly among attempts  $P_{max}/A$ .

We can construct an equation for the load as before,

$$L = Z + CA = \frac{Nf}{P'_{max}} + CA \quad (21)$$

which, by the definition of  $P'_{max}$ , implies that

$$L > A\left(\frac{Nf}{P_{max}} + C\right). \quad (22)$$

Substituting in Equation 20 for the number of attempts  $A$  per legitimate transaction yields

$$L > \left(\frac{L}{(L - F/T)q + F/T}\right)\left(\frac{Nf}{P_{max}} + C\right) \quad (23)$$

which simplifies to

$$(L - F/T)q + F/T > \frac{Nf}{P_{max}} + C \quad (24)$$

$$Lq > \frac{Nf}{P_{max}} + C + (q - 1)F/T \quad (25)$$

$$L > \frac{\frac{Nf}{P_{max}} + C + (q - 1)F/T}{q}. \quad (26)$$

Finally, we can find the probability of success  $p$  as a conditional probability  $P(win|stop)$ ,

$$p = \frac{\frac{F/T}{L}}{\frac{F/T}{L} + (1 - \frac{F/T}{L})q} \quad (27)$$

$$p = \frac{F/T}{(1 - q)F/T + Lq} \quad (28)$$

Substituting in the load (Equation 26) and simplifying, we can find the likelihood of a client succeeding in its transaction,

$$p < \frac{F/T}{(1 - q)F/T + \frac{Nf}{P_{max}} + C + (q - 1)F/T} \quad (29)$$

$$p < \frac{F/T}{\frac{Nf}{P_{max}} + C}. \quad (30)$$

Notice that the right side of this equation is the one-shot client case. Thus the optimum number of retries is one, and rather than allowing clients to retry until discouraged, the server should set its puzzle size so that they try precisely once and then abandon the transaction if unsuccessful.

Remember, though, that we have assumed homogeneous clients. The analysis and policy implications become markedly more complex for client populations with diverse patience or processing power because diversity introduces issues of fairness.

## VII. CONTRIBUTIONS

We have shown:

- what is required in implementation of a cookie puzzle mechanism to deny an attacker any remaining opportunities to amplify their attack,
- when attacks against resources protectable by a cookie puzzle mechanism dominate attacks against bandwidth,
- how much a cookie puzzle mechanism allows a server to protect its internal resources against denial of service attacks, and
- how to set difficulty for the puzzle under steady state attack.

Cookie puzzle mechanisms are an effective addition to the arsenal of DoS protection techniques, and are worthy of continued investigation and field testing. Although an attacker can still attack the server's processing power, the use of a cookie puzzle mechanism can vastly raise the amount of resources required for an attacker to reduce the server's capacity to service legitimate clients. Even for attackers with enough resources to disrupt service, the impact on legitimate clients is lessened, in some cases greatly.

Given homogeneous clients, dropping a client's transaction request is never advantageous from the perspective of server load or client service. The server should tune the puzzle difficulty to fit the attack precisely into its free capacity, and when that is not possible, set the puzzle difficulty to the maximum value tolerated by clients, thereby minimizing the number of client requests dropped.

The analysis and recommendations presented in this paper are limited to a steady-state situation. In order to be of practical use, however, the server must be able to cleanly transition between normal operation and various levels of attack, since needlessly difficult puzzles may have real costs for clients, such as battery drain or reduced performance.

A simple adaptive strategy such as exponential backoff may be employed to allow a server to adapt to changing circumstances. Attackers, however, may modulate their attack in an attempt to game the adaptive strategy. Further investigation is necessary.

An adaptive approach may also allow a server to distinguish between flash crowd events and DoS attacks, since a flash crowd event is a surge in user arrival rate and therefore non-responsive to increases in puzzle difficulty.

Another important direction for future investigation is the behavior of heterogeneous client populations, and the issues of fairness which arise. A server with a population which includes a significant portable (battery powered) contingent, for example, may have no reasonable puzzle difficulty settings to defend against an attacker with even a small collection of desktop zombies machines without unintentionally discriminating against its disadvantaged clients.

## REFERENCES

- [1] Jeff Ahrenholz, Tom Henderson, and Jeff Meegan. *Host Identity Protocol implementation for Linux*. The Boeing Company. Work in progress. <http://hipserver.mct.phantomworks.org/>
- [2] William Aiello, Steven Bellovin, Matt Blaze, Ran Canetti, John Ioannidis, Angelos D. Keromytis, and Omer Reingold. *Efficient, DoS-Resistant, Secure Key Exchange for Internet Protocols*. Proceedings of the ACM Computer and Communications Security (CCS) Conference. November 2002, Washington, DC.
- [3] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. *DoS-resistant authentication with client puzzles*. 8th International Workshop on Security Protocols, Cambridge, UK, April 2000. <http://citeseer.nj.nec.com/aura00dosresistant.html>
- [4] CERT. *Advisory CA-96.21: TCP SYN Flooding*, September 1996. <ftp://info.cert.org/pub/cert/advisories/CA-96.21.tcp-syn-flooding>
- [5] Drew Dean and Adam Stubblefield. *Using client puzzles to protect TLS*. In Proceedings of the 10th USENIX Security Symposium, Washington, DC., August 2001. <http://www.csl.sri.com/users/ddean/papers/usenix01b.pdf>
- [6] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. *Dynamic perfect hashing: upper and lower bounds*. SIAM J. Comput. 23 (1994), 738–761. <http://citeseer.ist.psu.edu/dietzfelbinger90dynamic.html>
- [7] Hani Jamjoom and K. G. Shin. *Persistent Dropping: An Efficient Control of Traffic Aggregates*. In the Proc. of SIGCOMM'03, Karlsruhe, Germany, August 25-29, 2003.
- [8] Charlie Kaufman. *Internet Key Exchange (IKEv2) Protocol*. Internet Draft, Internet Engineering Task Force, September 2004. Work in progress, version 17. <http://www.ietf.org/internet-drafts/draft-ietf-ipsec-ikev2-17.txt>
- [9] Miika Komu, et al. *HIPL: HIP for Linux* Work in progress. <http://gaijin.iki.fi/hipl/>
- [10] Andrew McGregor. *Pyhip* Work in progress. <http://www.sharemation.com/adm01bass/pyhip/>

- [11] Jelena Mirkovic, Janice Martin and Peter Reiher. *A Taxonomy of DDoS Attacks and DDoS Defense Mechanisms*. Technical Report 18, University of California, Los Angeles - Computer Science Department, 2002.  
[http://lasr.cs.ucla.edu/ddos/ucla\\_tech\\_report\\_020018.pdf](http://lasr.cs.ucla.edu/ddos/ucla_tech_report_020018.pdf)
- [12] Robert Moskowitz, Pekka Nikander, Petri Jokela, and Thomas Henderson. *Host Identity Protocol*, Internet Draft, Internet Engineering Task Force, June 2004. Work in progress.  
<http://www.ietf.org/internet-drafts/draft-ietf-hip-base-00.txt>
- [13] Robert Moskowitz and Pekka Nikander. *Host Identity Protocol Architecture*. Internet Draft, Internet Engineering Task Force, June 2004. Work in progress, version 6.  
<http://www.ietf.org/internet-drafts/draft-moskowitz-hip-arch-06.txt>
- [14] Various. *HIP for BSD Project*. Ericsson Research NomadicLab, Finland. Work in progress. <http://www.hip4inter.net/>