Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks

Ari Juels   John Brainard

RSA Laboratories
20 Crosby Drive
Bedford, MA 01730
{ari,jbrainard}@rsa.com

Abstract

We introduce a cryptographically based countermeasure against connection depletion attacks. Connection depletion is a denial-of-service attack in which an attacker seeks to initiate and leave unresolved a large number of connection requests to a server, exhausting its resources and rendering it incapable of servicing legitimate requests. TCP SYN flooding is a well-known example of such an attack. We introduce a countermeasure that we refer to as a client puzzle protocol. The basic idea is as follows. When a server comes under attack, it distributes small cryptographic puzzles to clients making service requests. To complete its request, a client must solve its puzzle correctly. In this paper, we describe the client puzzle protocol and its proper parameterization, and give a rigorous proof of its security characteristics.

1 Introduction

The Internet has fulfilled much of its early promise of enabling a single computer to service the requests of many millions of geographically dispersed users. In consequence, however, the Internet has introduced security concerns of a new magnitude, making a single computer potentially vulnerable to attack from many millions of sources. As a number of recent incidents have illustrated, even if a server is effectively protected against intrusive security breaches, it may still be vulnerable to a range of denial-of-service attacks, such as *connection depletion* attacks. A connection depletion attack is one in which the attacker seeks to initiate and leave unresolved a large number of connection (or service) requests to a server, exhausting its resources and rendering it incapable of servicing legitimate requests. The well-known TCP SYN flooding attack is one of the best publicized of these attacks. Other attacks in the same genre include so-called "e-mail bomb" attacks, in which many thousands of e-mail deliveries are directed at a single target, as well as attacks mounted using high volume read or write traffic via FTP connections with the aim of saturating storage space or bandwidth. Also potentially vulnerable to serious connection depletion attacks is the SSL (Secure Socket Layer) protocol [12].

In this paper, we present a new approach that we refer to as the *client puzzle* protocol, the aim of which is to defend against connection depletion attacks. The idea is quite simple. When there is no evidence of attack, a server accepts connection requests normally, that is, indiscriminately. When a server comes under attack, it accepts connections selectively. In particular, the server hands out to each client wishing to make a connection a unique *client puzzle*. A client puzzle is an quickly computable cryptographic problem formulated using the time, a server secret, and additional client request information. In order to have server resources allocated to it for a connection, the client must submit to the server a correct solution to the puzzle it has been given. Client puzzles are deployed in conjunction with conventional time-outs on server resources. Thus, while legitimate clients will experience only a small degradation in connection time when a server comes under attack, an attacker must have access to large computational resources to create an interruption in service. Cryptographic puzzles have been used in the literature for several related tasks, such as defending against junk e-mail [10], creating digital time capsules [19], and metering Web site usage [11].

1.1 TCP SYN flooding

The TCP SYN flooding attack serves as a good example to illustrate some of the issues surrounding connection depletion attacks, as well as some of the proposed defenses.

On the 12th September 1996, the New York Times [1] and other major newspapers, e.g., [21], reported that a hacker mounting a TCP SYN flooding attack had succeeded in crippling Panix, a major New York Internet service provider. Later that month, the Web site of the New York Times itself was the target of a similar attack [2], and other serious attacks followed, such as that against WebCom [7]. In response to the Panix attack, CERT [4] issued an advisory stating that automated tools for mounting the TCP SYN attack had been published by several underground magazines, and also warning of possible future attacks. The CERT report indicated that, "There is, as yet, no complete solution for this problem."

The TCP SYN flooding attack aims to exploit a weakness in the TCP connection establishment protocol whereby a connection may be left "half-open." The protocol normally proceeds as a three-way handshake. A client wishing to initiate a TCP connection with a server begins by sending the server a SYN message. The server replies to the client by sending a SYN-ACK message, and then prepares the connection by allocating buffer space. The client completes the protocol by responding with an ACK message. At this point, the connection is established, and service-specific data can be exchanged between client and server. Full details of the TCP protocol are available at [6]. To mount a TCP SYN flooding attack, the attacker initiates a large number of connections, but leaves them uncompleted. In other words, the attacker repeatedly fails to send the final ACK message which completes the connection. Since the server allocates buffer space for each incomplete connection, the attacker may exhaust server memory designated for TCP requests, causing the server to deny legitimate connection requests. A more detailed description of the attack may be found in [4] or [5].

Note that when used, as is common, without client-side certificates, the SSL protocol [12] is subject to a similar form of connection depletion attack. A connection depletion attack against SSL, however, is more effective when based on exhaustion of the computational, rather than the memory resources of a server. This is because the SSL protocol requires the server to perform a computationally expensive public-key-based decryption operation. An attacker may seek to exploit this feature and overload a server by initiating many invalid SSL connections. Connection depletion attacks against SSL are less well-studied than those against TCP, perhaps because SSL is a newer protocol.

A number of mechanisms have been proposed to defend against the TCP SYN attack. Most of these are based on one of three different approaches: time-out, random dropping, or "syncookies". The time-out approach discards a half-opened TCP connection after a short period of time if the ACK packet from the client has not been received. While easy to implement in existing TCP servers, this approach can be defeated by an attacker who sends the SYN packets at a rate fast enough to fill the connection buffer. Moreover, a short time-out may be problematic for legitimate users whose network connections have long latency. In the random dropping approach, half-open connections are discarded at random when the connection buffer reaches a certain percentage of its capacity. This prevents a complete denial of service, since the server buffer is never completely full. On the other hand, legitimate connections are as likely to be discarded as those of an attacker, so substantial degradation in service may result. This is particularly the case when the attacker is capable of sending service requests at a substantially higher rate than legitimate users, as is possible in such environments as an internal network.

To date, the most successful defense against TCP SYN flooding has been the use of so-called "syncookies." In the syncookie approach, for each client request i, the server sets the sequence number in its SYN/ACK message to a value V_i calculated by hashing various connection parameters with a secret value known only to the server. Only upon receiving an ACK containing V_i from the client making request i does the server allocate resources for the connection. The primary limitation of the syncookie approach is its assumption that an attacker performing IP spoofing will not receive the SYN/ACK message sent to the spoofed address, and therefore will not be able to provide the server with the value V_i. This assumption is not always correct, particularly on an internal network, such as an Ethernet, on which intercepting packets is relatively easy. The ISAKMP key management protocol, used in the IETF's IP Security standard, provides for the use of cookies to defend against TCP SYN attacks [14]. The details of this approach are described in, e.g., [3, 13].

1.2 Advantages of client puzzles

The approaches described above for defending against TCP SYN flooding, namely the syncookie and dropped connection schemes, can of course be applied to any underlying service protocol vulnerable to a connection depletion attack, including those mentioned above.

The client puzzle protocol has several advantages over other such defenses. Most important is its robustness in a stronger attack model than the standard approaches. Recall that the syncookie approach to defending against TCP SYN flooding (and other connection depletion attacks) presumes that an attacker cannot intercept messages sent to spoofed IP addresses. The client puzzle protocol requires no such assumption, and is therefore useful for defending against internal attacks in cases where the syncookie approach fails. Similarly, the client puzzle protocol is capable of handling attacks mounted at very high speeds, which the dropped connection approaches cannot always do effectively. The client puzzle protocol also allows for graceful degradation in service when an attack is mounted: the size of the puzzles can be increased as an attack becomes more severe. This makes the protocol quite flexible. The client puzzle protocol can either be built straightforwardly into the underlying service protocol it seeks to defend, or can be layered on top. It can even be used in conjunction with dropped connection or other approaches.

The principal disadvantage of the client puzzle protocol by comparison with the standard approaches is its requirement for special client-side software. The syncookie or dropped connection approaches merely require a modification to the underlying service protocol on the server side. The client puzzle approach requires that the client already have a program capable of solving a client puzzle. Such a program, however, can be built into a browser, made available as a plug-in, or distributed by any one of a variety of other means. For an internal environment, in which software is homogenous or centrally distributed, the requirement for special-purpose software does not pose a problem.

1.3 Organization of this paper

The remainder of this paper is organized as follows. Section 2 describes the attack model in which we formulate the client puzzle protocol, and provides notation. We give the details of the client puzzle protocol in Section 3. We explain and give an example of how to choose an appropriate server buffer size for our protocol in Section 4. We conclude in Section 5. Since the time required for a client to solve a client puzzle varies randomly, it is important to demonstrate that the client puzzle protocol is not subject to probabilistic forms of attack. We prove this in the appendix.

2 Attack Model

In this section, we describe the attack model in which we formulate the client puzzle protocol. We assume a network with clients $\{C_i\}$, a server Ser, and an adversary Ad. We consider any setting in which the adversary Ad seeks to mount a connection depletion attack against a client/server protocol $\mathbf{M}$. We assume that Ad seeks to exploit the protocol $\mathbf{M}$ so as to overload either the memory or computational resources of a server Ser. We make the following five assumptions about the adversary Ad. The first three of these assumptions state restrictions on the capability of the adversary required for our protocol to be useful, while the latter two describe adversarial attack capabilities to which our protocol is resistant.

Assumption 1 *Ad cannot modify packets sent from any C_i to Ser.*

Any attacker who can modify packets at will can mount a denial-of-service attack simply by corrupting packets. If Assumption 1 did not hold, then, an adversary would not need to attack a server by means of connection depletion. If we alter Assumption 1 slightly, though, and assume that the adversary can modify only a limited number of packets, our client puzzle protocol and the analyses in this paper are still germane.

Assumption 2 *Ad cannot significantly delay packets sent from any C_i to Ser.*

We make Assumption 2 for essentially the same reason as we make Assumption 1. In particular, if an adversary can delay packets arbitrarily, then she can mount a denial-of-service attack without having to overload the server. Again, some relaxation of this assumption is possible. The time-out parameters T_1, T_2, and T_3 in our client puzzle protocol as described below may be adjusted to accommodate possible adversarial delays.

Assumption 3 *Ad cannot saturate the server, the network, or any port.*

In order for an adversary to mount a connection depletion attack, she must be able to inject a large number of packets into a network. We assume, however, that the adversary cannot disable a server or a port simply by the sheer volume of her requests. In other words, we assume the adversary incapable of sending requests to the server so quickly that the server cannot effectively reject these requests. If the adversary could do

so, she would not need to resort to a connection depletion attack.

Assumption 4 *Ad* can perform IP spoofing. In particular, *Ad* can simulate any IP address of its choice in messages sent to the server.

As remarked above, IP spoofing is a component of many connection depletion attacks (as well as other denial-of-service attacks). Most attack models assume, as in Assumption 4, that an attacker can simulate an arbitrary IP address in its messages to the server. The client puzzle protocol is resistant to IP spoofing.

Assumption 5 *Ad* can read any messages sent to any IP address.

The syncookie approach fails under this assumption, as it means that access to cookies is no longer privileged. Unless packets are encrypted, however, Assumption 5 may be realizable by an adversary on a public network. It is certainly relevant in the case in which an adversary is mounting an attack on an internal network such as an Ethernet.

Note that an additional assumption behind our exploration of the client puzzle protocol, as indeed behind all of the defenses against connection depletion (TCP SYN flooding) attacks mentioned above, is the unavailability of a public-key infrastructure (PKI). If all entities in a network can authenticate one another by means of trusted certificates, then an effective basis already exists for mitigating or preventing connection depletion and other denial-of-service attacks. (This assumes of course, that anonymity is not desired on the part of network entities.)

3 The Client Puzzle Protocol

In this section we describe our solution, using client puzzles, for defending a protocol **M** (such as TCP or SSL) against connection depletion attacks. We describe our solution as a protocol which is layered independently on top of the protocol **M**. It is also easily possible, however, to integrate it directly into **M**.

The idea behind our defense protocol may be summarized as follows. In order to execute protocol **M** with the server, a client first asks whether the server is under attack. If the server is not under attack, it answers "no", and protocol **M** is executed normally. If the server is under attack, it answers "yes", and sends the client a puzzle to solve. In order to initiate the protocol **M** with the server, the client must solve the puzzle in a specified time interval. For an adversary *Ad* to mount a connection depletion attack against a server, *Ad* must solve a large number of puzzles in a short space of time – too many to handle with limited computational resources.

3.1 Protocol notation

Let M_i denote the ith execution of the protocol **M** on the server by a legitimate client or by the attacker. We denote by M_i^d the dth message sent in the ith execution of protocol **M**. To simplify our notation, we assume that the protocol **M** is client-initiated, e.g., the first message in the protocol is sent from the client to the server, and subsequent messages alternate direction. Hence, M_i^1 is sent from the client to the server, M_i^2 from the server to the client, and so forth. (Note that our scheme can be straightforwardly adapted, though, to the case where **M** is a server-initiated protocol.)

We require some additional notation. Let $z<i>$ denote the ith bit of a bitstring z, and let $z<i,j>$ denote the sequence of bits $z<i>, z<i+1>, \ldots, z<j>$. Let h denote a non-invertible hash function whose output is of length l. (Since we won't require that h be collision resistant, h might be a fast hash function like MD4 [17, 18], which is known to be vulnerable to collision searches, but as yet still resistant to inversion [9].) Let τ denote the current time according to the server clock. Let s be a secret seed of appropriate length (say, 128 bits) held by the server.

3.2 Client puzzle construction and protocol parameters

A client puzzle P_i consists of a number of independent, uniformly sized *sub-puzzles*. The reason for composing a puzzle of multiple sub-puzzles is to increase the difficulty for an attacker in guessing solutions. For example, a $(k+3)$-bit puzzle with one sub-puzzle requires the same average work factor for brute-force solution as a puzzle with eight different k-bit sub-puzzles. It is possible, though, to guess a solution to the former with probability $2^{-(k+3)}$, but only 2^{-8k} for the latter. The effect of having multiple sub-puzzles is apparent in the proofs in the appendix, particularly in Lemma 4.

We denote the jth sub-puzzle in P_i by $P_i[j]$. We let the variable m specify the number of sub-puzzles in a client puzzle. Hence a puzzle P_i consists of sub-puzzles $P_i[1], P_i[2], \ldots, P_i[m]$. A sub-puzzle $P_i[j]$ is constructed as follows. A bitstring $x_i[j]$ is computed as the hash of a set of service parameters and a server secret s. (This is described in more detail below.) This bitstring $x_i[j]$ is hashed in turn to yield a bitstring

$y_i[j] = h(x_i[j])$. The sub-puzzle consists of a portion $x_i[j] \langle k+1, l \rangle$ of the bitstring $x_i[j]$ along with the bitstring $y_i[j]$. The solution to the sub-puzzle $P_i[j]$ – which we may generally assume to be unique – consists of the missing bits of $x_i[j]$, i.e., the k-bit substring $x_i[j] \langle 1, k \rangle$. Thus the computational hardness of a sub-puzzle (under appropriate assumptions about the properties of h) is equivalent to the hardness of searching a space of size 2^k, where one search step requires the computation of a hash. The solution to the complete puzzle P_i consists of the m different k-bit solutions to all of the component sub-puzzles. Thus, k and m together represent a joint security parameter governing the difficulty of solving a client puzzle P_i. Throughout the remainder of this paper, we drop the subscript i from puzzle variables as convenient. The construction of a sub-puzzle $P[j]$ in a client puzzle P is depicted in Figure 1.

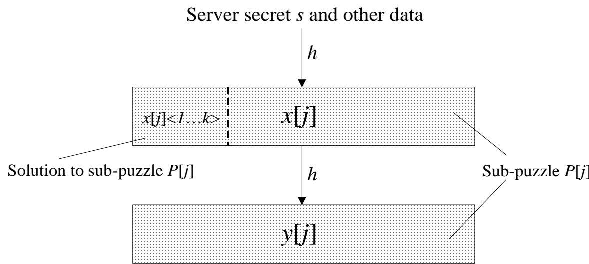

Figure 1. Constuction of sub-puzzle $P[j]$

We shall denote by B the size of the server *buffer* devoted to simultaneous executions of the protocol $\mathbf{M}$. We refer to as a *slot* the buffer resources devoted to a single execution of $\mathbf{M}$. Thus, the number of slots in B is the maximum number of simulataneous executions of $\mathbf{M}$ supported by the server. We let $maxcon$ denote the maximum number of simultaneous executions of $\mathbf{M}$ (i.e., connections) supported by the server during normal usage, i.e., the total number of slots in B reserved for normal usage with $\mathbf{M}$. Finally we let b denote the number of extra slots in B, i.e., the number of slots for executions of $\mathbf{M}$ anticipated during an attack, and thus set aside in support of the client puzzle protocol. Hence the total number of slots in B is $maxcon + b$. Note that the distinction between normal and extra slots is one of convenience and does not indicate any difference in functionality. The value b may be regarded as a defense parameter and represents in some sense the overhead associated with the client puzzle protocol.

As an example, if the protocol $\mathbf{M}$ is TCP, then the number of slots in B represents the amount of memory reserved for half-open connections. If the protocol $\mathbf{M}$ is SSL, then the number of slots in B represents the number of simultaneous SSL sessions (initiations) the server can support, and hence the computing power of the server reserved for such sessions. Hence the buffer B can represent either the memory or computational resource limitations of the server.

The client puzzle protocol includes three time parameters, T_1, T_2, and T_3. The parameter T_1 is the time for which a puzzle is valid before it expires, i.e., the time permitted to a client to solve and submit a client puzzle. The parameter T_2 is the time which the client has to initiate M_i after it has submitted a successfully solved puzzle. The parameter T_3 is the duration for which a buffer slot allocated for protocol M_i remains in memory before being purged. We let $T = T_1 + T_2 + T_3$.

3.3 Client puzzle protocol description

Let us now describe the details of the client puzzle protocol. Prior to initiating protocol $\mathbf{M}$, the client first sends message M_i^1 to the server, along with a query as to whether or not the server is distributing client puzzles. If the server is not under attack, it indicates to the client that no puzzles are being distributed, and records in memory, i.e., in a slot of B, the fact that initiation M_i of protocol $\mathbf{M}$ should be permitted. This permission is valid for a period of time T_2, after which it expires, and initiation of M_i is no longer permitted. If the client responds before expiration of the permission, it is free to execute initiation of M_i under $\mathbf{M}$ as normal. This execution must be completed in time T_3. The client puzzle protocol under normal server operation is depicted in Figure 2.

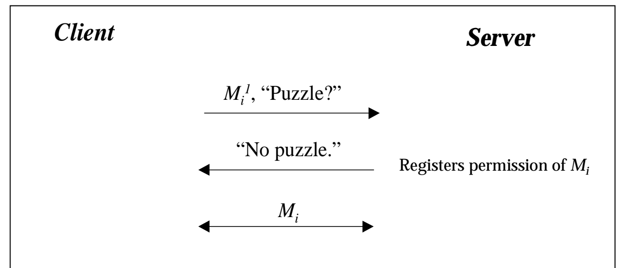

Figure 2. Protocol when server is operating normally

The server is deemed to be under attack if the server memory B begins to fill so that more than $maxcon$ buffer slots are allocated at one time. In this case, on requesting permission to initiate M_i, the client receives

from the server a client puzzle P. In order to have the server permit initiation of M_i, the client must submit a correct solution to P within time T_1. Once it has done so, the server sets aside a slot in B for M_i. In other words, the server records permission for the client to execute M_i, leaving the client free to proceed with the protocol $\mathbf{M}$ as normal.

Recall from above that a client puzzle P consists of a collection of m independent sub-puzzles. Each sub-puzzle $P[j]$ consists of a hash image with a partially revealed pre-image, while solving a sub-puzzle puzzle involves finding the remaining part of the pre-image. The client puzzle protocol requires that the server construct puzzles with two properties. First, the puzzle must be time-dependent, so that the client has only a limited time in which to solve the puzzle. Second, the puzzle must be able to be constructed in a stateless way. In particular, the server must be able to verify without the use of a database that a puzzle solved by a client is legitimate.

To satisfy these requirements, the server first generates a long-term, locally held secret s. This secret should be of length sufficient to forestall cryptanalytic attacks (e.g., 128 bits). When it receives a request from a client to initiate M_i, the server creates a puzzle P. For each sub-puzzle $P_i[j]$, the server computes the bitstring $x[j]$ as $h(s, t, M_i^1, j)$, where t is a timestamp set to the current time τ. (If M_i^1 is not unique to service requests, it may be desirable to include a nonce in the hash as well, this nonce effectively becoming an addition to the puzzle P.) Recall from above that the sub-puzzle $P[j]$ consists of the bits $x[j]{<}k+1, l{>}$ and the hash $y[j]$. The server sends the puzzle P to the client along with the timestamp t.

The client computes the solution to P by solving each sub-puzzle in turn. To solve a sub-puzzle, the client performs a brute force search of possible solutions. Searching a given solution will require the client to perform a single hash. We refer to the interval of time associated with this computation, i.e, the computation of a hash for trying a sub-puzzle solution, as a *time step*. A given k-bit sub-puzzle $P[j]$ will have 2^k possible solutions. Hence, the expected number of time steps for a client (or adversary) to solve a sub-puzzle will be $2^k/2 = 2^{k-1}$, while the maximum time will be 2^k. Since a puzzle contains m sub-puzzles, the expected number of time steps for a client (or adversary) to solve a puzzle P will be $m \times 2^{k-1}$, while the maximum will be $m \times 2^k$.

When the client sends t and M_i^1 along with its purported solution to P, the server can check in a stateless fashion that all sub-puzzles, and thus the entire puzzle P is correctly solved. It also checks, by comparing the timestamp t with the current time τ, that the puzzle has not expired. Hence the server need not store any puzzle information itself. In Figure 3, we denote by *solution* the (purported) solution data $(\{x[j]{<}1, k{>}\}_{j=1}^m, M_i^1, t)$. Figure 3 shows the client puzzle protocol when the server under attack. We assume in Figure 3 that a correct solution is submitted. If an incorrect solution is submitted, it will, of course, be rejected.

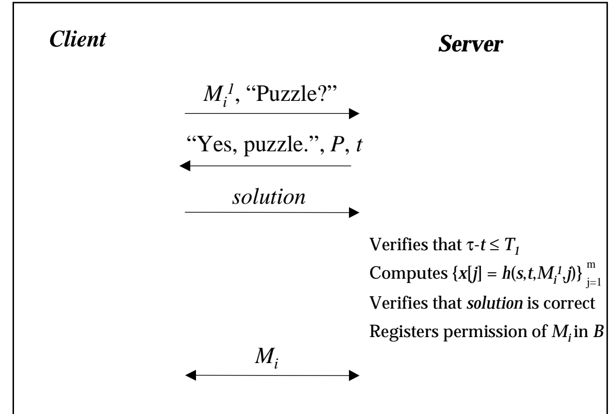

Figure 3. Protocol when server is under attack

Note that to prevent an attacker from using the same solved puzzle for multiple allocations, the server must ensure that only one slot in B is allocated for each request M_i. One way to accomplish this is to let some unique identifier derived from M_i^1 be associated with the slot allocated for M_i. On receiving a correctly solved puzzle corresponding to M_i, the server checks that no slot has been allocated for it already. One means of enabling a rapid search for already-used identifiers would be to assign slots through bucket hashing on identifiers.

Since a puzzle includes inversion problems of very limited size, and we are therefore not concerned with collision attacks, we can use relatively short hash images and pre-images. In determining the size of a sub-puzzle pre-image $x[j]$, we are principally concerned with the possibility of dictionary attacks. To avert such attacks, a 64-bit pre-image is sufficient in most settings. In considering the size of a sub-puzzle image $y[j]$, we wish to ensure, for the sake of our analyses, that the sub-puzzle is very likely to have only a single solution. We assume, in fact, in our proofs that this is the case. Under this constraint, it is reasonable likewise to make the sub-puzzle image 64 bits long. Given sub-puzzles of these proposed sizes, the size of a puzzle will be somewhat less than $16m$ bytes.

3.4 Puzzle efficiency: Improved puzzles

For the sake of simplicity, we assume in our exposition above that sub-puzzles are constructed independently of one another. If we allow some "overlap" in sub-puzzle construction, however, it is possible to construct puzzles that are substantially more compact and easy to verify, at the expense of some complexity and less easily demonstrable cryptographic properties. To construct a puzzle in this way, we allow the server to compute a value x of, say, $l/2$ bits derived from $h(s, t, M_i^1)$. The value x alone is sufficient to define m distinct sub-puzzles $P[1], P[2], \ldots, P[m]$ as follows. We let sub-puzzle $P[i]$ consist of the task of finding an $(l/2)$-bit value z_i such that the first k bits of $(x \parallel i \parallel z_i)$ match the first k bits of $h(x \parallel i \parallel z_i)$; here $\parallel$ denotes bitstring concatenation and i represents a fixed-length encoding of the sub-puzzle index i. An $(l/2)$-bit substring z_i with the desired property constitutes a correct solution to sub-puzzle $P[i]$.

To construct a puzzle of this sort, the server need only compute a single hash. As with the less efficient puzzle construction in which sub-puzzles are computed individually, we require only resistance to inversion and not collision-resistance in the hash function. Thus, it would still be suitable to use MD4. Since computing an MD4 hash requires only about 400 instructions, construction of a puzzle of this more efficient type requires only about 400 instructions on the part of the server. In a typical setting such as that described in Section 5, a puzzle would be roughly 8 bytes long (plus service values), while a puzzle solution would be about 64 bytes long.

Complete verification of a correct puzzle of this type requires $m + 1$ hash computations: one hash to compute x and another m to verify sub-puzzle solutions. On the other hand, if a puzzle is incorrect, it must contain at least one incorrectly solved sub-puzzle. Hence, by checking sub-puzzles in random order, the server can be assured of performing at most $m/2 + 1$ hash computations on average to verify an incorrectly solved puzzle. In a typical setting, such as that described in Section 5, in which $m = 8$, this would correspond to about 2,000 instructions.

Alternative embodiments of this puzzle construction are achievable using a fast block cipher, e.g., RC6 [16], in lieu of a hash function. In fact, it would be acceptable to use a reduced-round block cipher for this purpose, provided that it is still not worthwhile for an attacker to mount a cryptanalytic attack. Finding more efficient puzzle constructions represents an interesting research problem.

3.5 Graceful degradation

In the simple implementation of the client puzzle protocol described above, the security parameters k and m are fixed. In other words, when the server comes under attack (i.e., *maxcon* buffer slots are allocated in B), it distributes puzzles of uniform difficulty to clients requesting initiation of protocol **M**. Note, though, that it is possible for the server to scale puzzle sizes and thus impose variably sized computational loads on the client. In particular, a client puzzle protocol might scale the hardness of puzzles according to the severity of the attack on the server: the more the server buffer becomes filled, the harder the puzzles it issues. It is also to make changes to the time-out parameters T_1, T_2, and T_3. Through modification of client puzzle parameters, server performance can be caused to degrade gracefully, and in proportion to the severity of the threat to the server.

4 Choosing a Correct Size for the Buffer B

Recall that *maxcon* is the number of connections made available to clients by the server under normal operating conditions. When implementing the client puzzle defense against connection depletion attacks, we let the buffer B contain $maxcon + b$ slots. Our purpose in this section is to determine an appropriate size for b. Recall that, on average, it will take an expected $m2^{k-1}$ time steps for the adversary to solve a puzzle. Hence the expected number of time steps for the adversary to solve b puzzles is $bm2^{k-1}$. If the buffer B contains $maxcon + b$ slots, then the adversary must solve b puzzles in T seconds to mount a successful connection depletion attack, where, again, $T = T_1 + T_2 + T_3$. Let g represent the number of adversarial time steps per second, i.e., the number of hashes the adversary can compute per second. Thus Tg represents the number of time steps the adversary has to carry out its attack. Since the expected number of time steps for the adversary to solve b puzzles is $bm2^{k-1}$, we would expect that $b > Tg/(m2^{k-1})$ would be likely to foil the adversary most of the time.

The problem with this reasoning is that it does not take into account some complicating factors. First of all, it is possible that if the adversary mounts a sustained attack, she will get lucky at some point and encounter a succession of easy puzzles. In addition, a crafty adversary may not explicitly solve all of the puzzles she submits. Instead, she may try to guess the solutions to some puzzles, or attack puzzles by partially solving and partially guessing at them. We

therefore require rigorous proof to establish the size of the buffer required for a successful client puzzle protocol. Let us define $c = bm2^{k-1}/Tg$. Recall that the expected time for the adversary to solve b puzzles is $bm2^{k-1}$. Intuitively, therefore, c expresses roughly the number of times larger the value b is than we would expect to need. We have the following theorem, whose proof is given in the Appendix. Note that 10^8 MIPS represents an arbitrary upper bound on the power of the adversary, and is given only to simplify the form of the theorem.

Theorem 1 *Assume an adversary with at most* 10^8 *MIPS of computing power that mounts a connection depletion attack for no more than a year. Assume further that puzzles are constructed such that* $m \geq 8$. *Then if* $c \geq 7/2$ *and* $b \geq 1100$, *the probability that the adversary is able to mount a successful attack is less than* 2^{-100}. □

Note that the size of k is determined by c and by the parameter T. In order to simplify our analysis, Theorem 1 is proven for parameters larger than would be needed in practice, i.e., with a rather conservative approach to security. In practice, designers are encouraged instead to make use of the following heuristic. Again, the upper bound of 10^8 MIPS is selected arbitrarily to simplify the form of the theorem.

Heuristic 1 *Assume an adversary with at most* 10^8 *MIPS of computing power that mounts a connection depletion attack for no more than a year. Assume further that puzzles are constructed such that* $m \geq 8$. *Then if* $c \geq 4/3$ *and* $b \geq 1000$, *the probability that the adversary is able to mount a successful attack is less than* 2^{-100}. □

Note that Theorem 1 and Heuristic 1 are quite crude: for the sake of simplicity, they rely on a number of assumptions which can be relaxed somewhat in a real system design. For example, sizes of m larger than 8 reduce the required buffer size. By appealing to Corollary 2, as given in the Appendix, a protocol designer can obtain tighter bounds on buffer sizes.

5 An Example

As a demonstration of how the client puzzle protocol ought to be parameterized, let us consider a scenario in which an attacker wishes to mount a TCP SYN flooding attack. Suppose the system under consideration is one in which clients typically have 100 MIPS of computing power, and an adversary is anticipated to be able to harness the full power of 20 client machines. Thus the adversary has 2,000 MIPS of computing power at her disposal.

Let us assume that MD4 is the hash function used to construct puzzles. One MD4 computation requires about 400 instructions. Hence, a client can perform 250,000 MD4 computations/second. The number of time steps per second g for the adversary, which we assume to have 2,000 MIPS of computing power, is 5×10^6. The DEC archive note "Performance Tuning Tips for Digital Unix" [8] suggests that half-open TCP SYN connections be permitted to persist for a period of time on the order of 75 seconds, i.e., $T_3 = 75$ seconds. Let us assume that $T_1 + T_2$ is also about 75 seconds. In other words, a client has 75 seconds to receive and submit a solved puzzle and initiate a TCP connection. Hence $T = 150$ seconds. This corresponds to $150 \times 5 \times 10^6 = 7.5 \times 10^8$ time steps for the adversary.

Suppose we client puzzles consisting of 8 sub-puzzles, each with 16 bits. Thus, $m = 8$ and $k = 16$. The search space associated with a puzzle is of size $8 \times 2^{16} = 524,288$. Hence, on average, a client should take just under a second to solve a puzzle.

A server in the 100 MIPS class has a typical maximum processing speed of between 1,000 and 2,000 TCP connections per second [20]. Thus a back-of-the-envelope calculation reveals that the roughly 400 instruction overhead associated with construction of a puzzle (see section 3.4) translates into a performance penalty of less than 1% relative to TCP connection computation time. Verification of an incorrect puzzle takes at most an average of 2,000 instructions, equivalent to at most 4% of the time required to establish a TCP connection. As noted in Section 3.4, puzzles can be constructed to be about 8 bytes in size (plus service information) while solutions are about 64 bytes in size. These penalties are small and, of course, paid only when the server is under attack.

Now, by Theorem 1, we require at least $c \geq 7/2$ and $b \geq 1100$. Since $b = cTg/(m2^{k-1})$, this means that we require $b \geq (7/2 \times 7.5 \times 10^8)/(8 \times 2^{15}) \approx 10,000$. Hence we need B to contain about $10,000 + maxcon$ buffer slots. Heuristic 1, however, specifies that $c \geq 4/3$ and $b \geq 1000$. This instead gives us $b \geq (4/3 \times 7.5 \times 10^8)/(8 \times 2^{15}) \approx 3,750$. Hence, in practice, we would want B to contain on the order of $3,750 + maxcon$ buffer slots. Again, by appealing to Corollary 2, as given in the Appendix, more careful analysis would yield a smaller value of b.

In conventional environments, it is standard to adopt a minimum server TCP buffer size of 2048 slots [8]. This includes buffer slots devoted to defense against TCP SYN flooding attacks. Hence the client puzzle protocol does not yield a substantially larger

buffer size than current recommendations.

Of course, the example above is based on a simple client puzzle protocol in which puzzles are of uniform difficulty, requiring just over a second for a client machine. Graceful degradation would lead to a better client puzzle parameterization, and consequently to a smaller requirement on the server buffer size.

6 Conclusion

We have presented a protocol in which small cryptographic puzzles may serve as a deterrent against connection depletion attacks. This protocol is especially valuable against attackers capable of depleting connections at a fast rate, such as attackers on internal networks or attackers exploiting resource-intensive connection protocols like SSL.

There is a great deal of room for further exploration of the techniques we have presented here. Through a more detailed consideration of the properties of hash functions and the time/space tradeoffs faced by an attacker, for example, it may be possible to develop a form of more compact and/or more efficiently computable client puzzle. Further investigation might also yield better theoretical results, perhaps narrowing the gap between the protocol settings suggested in our main theorem (Theorem 1) and those which we suggest in practice (Heuristic 1).

Acknowledgements

The authors wish to extend thanks to Ulana Legedza for bringing the connection depletion problem to their attention. We also wish to thank Mor Harchol-Balter, Markus Jakobsson, Ronald Juels, Burt Kaliski, and John Linn for helpful discussions regarding the ideas in this paper, as well as the anonymous referees for their detailed comments.

References

[1] *The New York Times*, 12 September 1996.

[2] R. Aguilar and J. Kornblum. *New York Times* site hacked. *CNET NEWS.COM*, 8 November 1996.

[3] R. Braden. T/TCP: TCP extensions for transactions functional specification, July 1994. RFC 1644, Internet Activities Board.

[4] CERT. *Advisory CA-96.21: TCP SYN flooding and IP spoofing attacks*, 24 September 1996.

[5] Daemon9. Project Neptune. *Phrack Magazine*, 48(7):File 13 of 18, 8 November 1996. Available at www.fc.net/phrack/files/p48/p48-13.html.

[6] DARPA. *Transmission control protocol (TCP) DARPA Internet program protocol specification*, September 1981. Available at http://ds.internic.net/rfc/rfc793.txt.

[7] C. Davidson. The "SYN flood" gates open for WebCom. *iWorld Weekly*, 16 December 1996.

[8] Digital Equipment Corporation. *Performance Tuning Tips for Digital Unix*, June 1996. Available at www.service.digital.com/manual/misc/perf-dec.html.

[9] H. Dobbertin. Cryptanalysis of MD4. In D. Grollman, editor, *Fast Software Encryption: Third International Workshop*, pages 53–69. Springer-Verlag, 1996. Lecture Notes in Computer Science No. 1039.

[10] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Proc. CRYPTO 92*, pages 139–147. Springer-Verlag, 1992. Lecture Notes in Computer Science No. 740.

[11] M.K. Franklin and D. Malkhi. Auditable metering with lightweight security. In R. Hirschfeld, editor, *Proc. Financial Cryptography 97 (FC 97)*, pages 151–160. Springer-Verlag, 1997. Lecture Notes in Computer Science No. 1318.

[12] A. Freier, P. Karlton, and P. Kocher. The SSL protocol - version 3.0, March 1996. Available at http://home.netscape.com/eng/ssl3/ssl-toc.html.

[13] P. Karn and B. Simpson. Photuris: Session key management protocol. I-D draft-ietf-ipsec-photuris-08.txt, work in progress, February 1998.

[14] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet security association and key management protocol (ISAKMP). Technical report, IETF, March 1998. Internet-Draft:draft-ietf-ipsec-isakmp-09.txt.

[15] C. McDiarmid. On the method of bounded differences. In *Survey in Combinatorics*, pages 148–188. Cambridge University Press, 1989. London Math. Soc. Lecture Note Series 141.

[16] R. Rivest, M. Robshaw, R. Sidney, and L. Yin. The RC6 block cipher, 1998. Presented at the NIST First AES Candidate Conference, Ventura, CA.

[17] R.L. Rivest. The MD4 message digest algorithm. In A.J. Menezes and S.A. Vanstone, editors, *Proc. CRYPTO 90*, pages 303–311. Springer-Verlag, 1991. Lecture Notes in Computer Science No. 537.

[18] R.L. Rivest. The MD4 Message-Digest Algorithm, April 1992. RFC 1320, Internet Activities Board.

[19] R.L. Rivest, A. Shamir, and D. Wagner. Time-lock puzzles and timed-release crypto. Manuscript, To appear, 10 March 1996.

[20] The Standard Performance Evaluation Corporation. *SpecWeb96 Benchmark Results*, 1998. Available at http://www.specbench.org/osg/web96/results.

[21] B. Ziegler. Hacker tangles Panix Web site. *Wall Street Journal*, 12 September 1996.

A Appendix: Proofs

In this appendix, we analyze and prove results about the level of defense provided by the client puzzle protocol. At first glance, this would seem simple, as we can easily compute the average time required by the adversary to solve a puzzle, and we know the number of puzzles required to overload the server. Two facts, however, make the analysis tricky. The first is that the time required to solve a client puzzle is a random variable. Since solving a client puzzle involves a search of the space of potential solutions, it is possible that an adversary may, with luck, happen upon the solutions to a large number of puzzles in a short space of time. The second fact is that it is possible for an adversary to guess at the solution to a client puzzle. With some small probability, an adversary's guess will be correct, and the server will allocate buffer memory for a connection. Together, these two facts mean that even an adversary with limited computing resources can mount a successful attack against the client puzzle protocol with some probability. By mixing the processes of solving and guessing appropriately, it might in principal be possible for a clever adversary to mount a successful attack. Our aim in this section is to prove that when the client puzzle protocol is correctly parameterized, the probability of mounting a successful attack in a reasonable space of time is negligible for any adversary.

A.1 Preliminaries: notation and assumptions

Let P be a puzzle of the form described above, and let $P[j]$ be the jth sub-puzzle of P. Let $S[j]$ be the set of possible solutions to $P[j]$, i.e., the 2^k possible assignments to bits $x[j]<1,k>$. Since $x[j]$ and $y[j]$ are of equal length, we assume in our analysis that h, the hash function used to generate the sub-puzzles in P, is a random permutation. (In other words, we assume that h is 1-1 when input and output are of equal length, and we also make a random oracle assumption on h.) Thus a sub-puzzle has a unique solution, and the most effective strategy for solving a sub-puzzle $P[j]$ is brute force, i.e., an arbitrarily ordered (but non-redundant) search of the space $S[j]$. As above, we define a time step to be the examination of a single element of $S[j]$. Recall that the expected number of time steps for a brute force search to yield a solution to $P[j]$ is 2^{k-1}, and the expected number of time steps for a brute force search to yield a solution to P is $m2^{k-1}$. We assume that the time for the adversary to guess at and submit a solution to a puzzle is one time step. As the application of a hash function such as MD4 requires about 400 instructions, this assumption is accurate (and perhaps even conservative) for real-world settings.

Let us denote by S the set of possible solutions to the puzzle P. Let us use $\times$ to denote the product of sets. The set S of possible solutions to P consists of solutions to all of the sub-puzzles in P, i.e., $S = S[1] \times S[2] \times \ldots \times S[m]$. Hence the set S is isomorphic to the set of all bitstrings of length km. Let S^* denote the subset of S containing all solutions to P which the adversary has not yet eliminated from its search.[1] Suppose, for example, that the adversary has solved sub-puzzle $P[1]$, and has unsuccessfully searched a set consisting of half of the possible solutions to sub-puzzle $P[2]$. Then $S^* = x_1 < 1 \ldots k > \times (S[2] - S^*[2]) \times S[3] \times \ldots \times S[m]$, so S^* effectively contains 2^{km-k-1} bitstrings. Let us define $s(P) = m - \sum_j (|S^*[j]|/|S[j]|)$. Hence, in our example here, $s(P) = 3/2$. The quantity $s(P)$, which we refer to as the *solution level* of P, may be loosely regarded as the number of sub-puzzles which the adversary has solved in P. We may define $s(P[j])$ analogously to be the portion of sub-puzzle $P[j]$ which the adversary has solved. In particular, $s(P[j]) = 1 - (|S^*[j]|/|S[j]|)$.

In the client puzzle protocol as described above, there are three expiration times: expiration time T_1 on client puzzles, T_2 on permission to initiate protocol execution M_i, and T_3 on the buffer associated with M_i. An attacker seeks to solve as many client puzzles as possible subject to these constraints. Let $T = T_1 + T_2 + T_3$. It is easy to see that at a given point in time τ, there will correspond to each allocated buffer slot in B a solved client puzzle which was requested at the latest at time $\tau - T$. Therefore, the maximum number of buffer slots an adversary can have allocated in B is bounded above by the number of client puzzles she can solve in time T. If b buffer slots in B are devoted to defending against an adversary, the probability of success of attack of an adversary is bounded above probability that she can solve b puzzles in an interval of time T. We make the assumption that the adversary can request and receive solved puzzles instantaneously, and also submit solved puzzles instantaneously. Other, more important bounding assumptions are enumerated below.

Let us consider a scenario in which an entity seeks to perform a computation C in discrete time steps, starting with time step $t = 1$. We de-

[1] Note that when $|S^*| = 1$, the adversary has solved the puzzle in question, so that $|S^*| = 1$ and $|S^*| = 0$ are, for all intents and purposes, equivalent conditions. We let either $|S^*| = 1$ or $|S^*| = 0$ denote a finished puzzle - whichever is convenient for notation and mathematical consistency in a given context. We do likewise for $S^*[j]$ for any sub-puzzle.

fine an *optimal* algorithm or strategy for an performing computation C in U time steps as follows. Let c_t be the state of the computation in time step t, and let $\text{pr}[(X, U, C)]$ be the probability that algorithm X completes computation C by time U. Algorithm X is optimal if for all algorithms Y and times $t \leq U$, $\text{pr}[(X, U, C) \mid c_t, c_{t-1}, \ldots, c_1] \geq \text{pr}[(Y, U, C) \mid c_t, c_{t-1}, \ldots c_1]$. In other words, employing algorithm X yields the highest probability of success at any time irrespective of the history of the computation.

Finally, we let BF denote the brute force strategy for solving sub-puzzles. In this strategy, the adversary works on the (or an) unsolved sub-puzzle SP such that $s(SP)$ is maximal. In particular, the adversary searches an arbitrary, unexamined solution to SP. Observe that if the adversary employs BF from the start, then BF means solving sub-puzzles in strict sequential order. In other words, the adversary requests a sub-puzzle, works on it until it is solved, and only then requests another puzzle.

A.2 Bounding assumptions

We consider an adversary with limited computational resources seeking to attack a server over some interval of time U. The adversary can submit attempted puzzle solutions with variable solution levels. The adversary can, for instance, perform no computational effort, submit a random solution to P with solution level $s(P) = 0$, and hope that it has guessed correctly. With this strategy, the adversary can submit many potential solutions in a short space of time, but few of these solutions are likely to be correct. Alternatively, the adversary may invest the full computational effort to solve puzzles completely, i.e., submit puzzles with solution level $s(P) = m$. With this strategy, the adversary is assured that its submitted solution to a given puzzle P is correct, but it can submit solutions to relatively few puzzles within a given space of time. There is, of course, also a continuum of intermediate levels of computational investment the adversary may make in submitting attempted puzzle solutions. For the purposes of simplifying our proof, we divide this continuum into two parts. Those solutions that the adversary submits to puzzles P with $s(P) \leq m/2$ we refer to as *short* solutions. Solutions to such puzzles are short in the sense that the portion of the solution that is computed, rather than guessed at, is short. In contrast, we refer to solutions such that $s(P) > m/2$ as *long* solutions. Such puzzles have a high solution level, i.e., most of the attempted solution has been worked out, rather than guessed at. Of course, we could partition the continuum of solution levels on a puzzle into more than two parts, but the classification into short and long solutions is sufficient for the purposes of our proof. Given this classification, we make use of the following bounding assumption to achieve an upper bound on the power of the adversary. This assumption is implicit throughout our proof.

Bounding assumption 1 If the adversary submits a short solution to a puzzle P (i.e., $s(P) \leq m/2$), then we assume $s(P) = m/2$. (Thus, even if the adversary invests no computation in a solution, we assume that it has correctly solved $m/2$ sub-puzzles.) If the adversary submits a long solution to a puzzle P (i.e., $s(P) > m/2$), then we assume $s(P) = m$, i.e., that the adversary has solved the puzzle completely.

To simplify our proof further, we make two additional bounding assumptions. These assumptions likewise have the effect of yielding an upper bound on the power of on an adversary mounting an attack over a time interval of length U. We apply this next bounding assumption to long solutions, simplifying the task of the adversary in submitting such solutions.

Bounding assumption 2 We relax the requirement that sub-puzzles be grouped in puzzles. Instead, when the adversary must submit X long solutions in a given time interval, we assume that it is sufficient for the adversary simply to submit $mX/2$ correctly solved sub-puzzles of its choice.

Normally, to mount a successful attack over a period of time U against a buffer B, the adversary must submit at least b correctly solved puzzles. We partition the b slots devoted to the client puzzle protocol in buffer B into two buffers B_1 and B_2 with (arbitrary) sizes $b/8$ and $7b/8$ respectively. We then render the task of the adversary easier as follows.

Bounding assumption 3 We allow the adversary to attempt the following two tasks, assuming that the adversary is successful if it performs either one successfully: (1) Fill buffer B_1 in time U by submitting enough correct short solutions, or (2) Fill buffer B_2 in time U by submitting enough long solutions. (Note that our use of Bounding Assumption 2 is such that we don't require long solutions to be correct to cause allocations in buffer B_2.)

A.3 Proof outline

To summarize, then, we allow the adversary time U to try to fill buffer B_1 by means of short solutions and

time U to try to fill buffer B_2 by means of long solutions. We show that short solutions are unlikely to be correct (Lemma 1). This leads us to show that the adversary has a very low probability of successfully filling buffer B_1 (Lemma 4). For the adversary to fill buffer B_2 successfully, we relax the requirement that the adversary submit the required number of correct long solutions. Instead, under Bounding Assumption 2, we allow the adversary to submit an appropriate number of independent sub-puzzles. An adversary may seek to exploit variations in solution time of sub-puzzles – perhaps by means of a probabilistic strategy – to reduce the computational power it requires to mount a successful attack. We show, however, that the brute force algorithm (BF) is optimal for the adversary in solving independent sub-puzzles (Lemma 2). We show further that the time for BF to solve the necessary number of sub-puzzles is long enough so that it is very unlikely the adversary can do so in time U (Lemma 5). Since the adversary is likely to be able to fill neither buffer B_1 nor buffer B_2, we obtain our main theorem, Theorem 1, stating that the adversary is very unlikely to be able to mount a successful denial of service attack by overloading buffer B in a feasible interval of time U.

A.4 Proofs

We begin by showing that short solutions are, in general, unlikely to be correct.

Lemma 1 *The probability that a short solution is correct is* $\leq 2^{-km/2}$.

Proof: Suppose that $s(P[j]) + s(P[j']) < 2$. In other words, suppose that the adversary has not solved sub-puzzles $P[j]$ and $P[j']$ completely. Let q be the probability that if the adversary guesses at a solution to P, it will guess correct solutions to $P[j]$ and $P[j']$. It is straightforward to see that $q = 1/(|S^*(P[j])||S^*(P[j'])|)$. If $s(P[j]) + s(P[j']) < 1$, then $|S^*(P[j])| + |S^*(P[j'])| > 2^k$. It is easy to show that q is maximized when $|S^*(P[j'])| = 2^k$ and $|S^*(P[j])|$ is minimized, i.e., when $s(P[j]) = s(P[j]) + s(P[j'])$, and $s(P[j']) = 0$. If $s(P[j]) + s(P[j']) \geq 1$, then q is similarly maximized when $|S^*(P[j])| = 1$, i.e., when $s(P[j]) = 1$. All of this is to say, in loose terms, that q is maximized when the search of the adversary has been concentrated as much as possible on one sub-puzzle. By a straightforward inductive argument, it follows that the probability that an adversary makes a correct guess on a puzzle P is maximized when the search of the adversary has been concentrated on as few sub-puzzles as possible. Call this probability r. The probability r is maximized when $s(P[1]) = s(P[2]) = \ldots = s(P[\lfloor s(P) \rfloor]) = 1$ and $s(P[\lceil s(P) \rceil] = s(P) - \lfloor s(P) \rfloor)$. It is easy to show now that for any short solution, $r \leq 2^{-k(m-s(P))}$. The Lemma follows. □

Observation 1 *Let SP and SP' be two sub-puzzles. Let p be the probability that the adversary solves SP on the next step of work on that sub-puzzle, and p' be the probability that the adversary solves SP' on the next step of work on SP'. If $s(SP) \leq s(SP')$, then $p \leq p'$.*

Proof: The probability that an adversary solves a unsolved sub-puzzle SP by working on it for one time step is $1/(|S^*(SP)|)$, and thus increasing with respect to $s(SP)$. The observation follows. □

Observation 1 means that the work the adversary works on a sub-puzzle, the more it pays to continue to work on that sub-puzzle. This observation is crucial to our proofs, suggesting as it does that the optimal strategy for the adversary is to work on puzzles in a strictly sequential fashion. We proceed to prove that this is the case.

Lemma 2 *Let $SP_1, SP_2, \ldots$ be a sequence of (partially solved) sub-puzzles, and $z \geq 0$. Then for any n, BF is an optimal algorithm for an adversary to achieve $\sum_i s(SP_i) \geq z$ in n time steps.*

Proof: Let n be an arbitrary number of time steps, and let C denote computation achieving $\sum_i s(SP_i) \geq z$. Let H be the history of computation of all computation performed by the adversary on puzzles $SP_1, SP_2, \ldots$. Let c be the current state of computation. We shall prove inductively on j that for all Y, it is the case that $\text{pr}[(BF, j, C) \mid H] \geq \text{pr}[(Y, j, C) \mid H]$. In other words, BF is optimal. Consider $j = 1$. If the state of the computation c at this point is such that $\sum_i s(SP_i) \geq z - 1/2^k$, then any algorithm which works on an unsolved sub-puzzle will complete computation C. If $\sum_i s(SP_i) < z - 1/2^k$, then either C cannot be achieved, or else the only way that C can be achieved by time step n is for the adversary to solve an unsolved sub-puzzle in the next time step. Let us assume the latter. By Observation 1, the adversary maximizes its probability of achieving this by working on the unsolved sub-puzzle SP_i such that $s(SP_i)$ is maximal. In other words, BF is optimal.

Let us assume inductively that BF is optimal for $j < n$, and now consider $j = n$. Let us refer to the first time step at this point as the initial time step. Let us denote by $W(1)$ those unsolved sub-puzzles which have received the maximal amount of work prior to the initial time step, i.e., all unsolved sub-puzzles

SP such that $s(SP)$ is maximal among unsolved sub-puzzles. Let us denote by $W(2)$ those unsolved sub-puzzles which have received the second largest quantity of work, etc. The algorithm BF would, in this initial time step, work on some sub-puzzle in $W(1)$. Let us suppose instead that there is an optimal algorithm X which works on some sub-puzzle which has received less than the maximal amount of work, i.e., some sub-puzzle $SP' \in W(z)$ for $z > 1$. This, of course, either promotes SP' to $W(z-1)$ or solves it. Let us regard the partitioning of sub-puzzles defined by W as frozen from this point on.

By induction, BF will be optimal after this time step, so we may assume that X runs BF for all of the $n-1$ remaining time steps. This means that the adversary is working on sub-puzzles in $W(1)$, then sub-puzzles in $W(2)$, etc. We can examine the progress of the adversary in these remaining time steps in terms of two cases.

Case 1: The adversary never finishes $W(z-1)$, i.e., doesn't solve all sub-puzzles in $W(z-1)$. In this case, the adversary applies only one step of work to SP'. If the adversary had not worked on SP' in the initial time step, and had pursued algorithm BF instead of X, it would have been instead able to devote one more step of work to some puzzle SP in $W(z-i)$ for some $i \geq 1$. In other words, the adversary could have worked on some puzzle SP such that $s(SP) \geq s(SP')$. By Observation 1, this would have meant at least as high a probability of success of completing computation C. Hence employing BF in the initial time step yields at least as high a probability of success as employing X.

Case 2: The adversary finishes $W(z-1)$. In this case, the adversary has merely shuffled the order of its work. The adversary has applied one initial step of work to SP', either solving SP' or promoting it to $W(z-1)$. If the adversary promoted SP' to $W(z-1)$, then it subsequently applied w additional steps of work to complete SP'. In either case, if the adversary had instead not worked on SP' in the initial time step, leaving SP' in $W(z)$, it would have reserved at least one extra step until after it had completed $W(z-1)$. In this case it would have applied at least $w+1$ steps to a sub-puzzle in $W(z)$, which we may assume, w.l.o.g. is SP'. Hence, in either case, SP' would have been solved (with the same amount of work, of course). Hence employing BF in the initial time step yields at least as high a probability of success as employing X. □

The following corollary is now immediate.

Corollary 1 *The optimal algorithm for computing a long solution to puzzle P is to apply BF to its component sub-puzzles until $m/2$ sub-puzzles are solved.* □

Corollary 1 would seem to imply that an optimal adversarial strategy for computing as many long solutions as possible in a given interval of time is to work on puzzles sequentially. This is not necessarily the case, however. Corollary 1 tells us that an optimal adversary will apply BF to the sub-puzzles in any given puzzle. A crafty adversary, however, might increase its efficiency by interleaving work on a number of puzzles, and concentrating more work on those puzzles on which it is making the best progress. Rather than attempting to analyze an adversary of this sort, we make use of the following, simple observation.

Observation 2 *In order to submit z long solutions, an optimal adversary must correctly solve at least $zm/2$ sub-puzzles.* □

Observation 2 enables us to invoke Bounding Assumption 2 from above. Now, instead of requiring the adversary to submit z long solutions in a given space of time, we allow the adversary instead to submit $zm/2$ sub-puzzles. We permit the adversary to request and solve these sub-puzzles independently of one another.

As noted above, in order for an adversary to have b buffer slots allocated at a given time t, the adversary must have submitted correct solutions to at least b puzzles requested in the previous time interval T. Some of these solutions may be short, and some may be long. We bound the power of the adversary separately with respect to short and long. To do this, we invoke Bounding Assumption 3.

Recall that under Bounding Assumption 3, we partition the b slots in buffer B devoted to the client puzzle protocol into memory spaces B_1 and B_2. Buffer memory B_1 is reserved for the submission of correct short solutions. We allocate to B_1 (somewhat arbitrarily) a $1/8$-fraction of the buffer slots b in B, i.e., the number of buffer slots in B_1 is equal to $b/8$. Buffer memory B_2 is reserved for the submission of long solutions. We allocate to B_2 the remaining $7b/8$ buffer slots. In order to mount a successful attack, the adversary must overload either buffer B_1 or buffer B_2 (or both). We show that an adversary is unlikely to be able to do either.

We first require the following lemma, based on the Hoeffding inequalities presented in McDiarmid [15].

Lemma 3 (Hoeffding Inequalities) *Let $X_1, X_2, \ldots, X_n$ be a set of independent, identically distributed random variables in [0,1], and let $X = \sum_i X_i$. Then:*

1. $pr[X - E[X] \geq \epsilon E[X]] \leq exp(-1/3\epsilon^2 E[X])$ *and*
2. $pr[X - E[X] \leq -\epsilon E[X]] \leq exp(-1/2\epsilon^2 E[X])$.

□

Let us first turn our attention to buffer B_1, the buffer devoted to correct short solutions. The following lemma states that if b is sufficiently large, then p_{B_1}, the probability that the adversary mounts a successful attack against B_1, is small. In accordance with our notation above, let us denote by g the number of time steps per second for the adversary, i.e., the number of hashes per second the adversary can perform.

Lemma 4 *Let us denote by p_{B_1} the probability that the adversary mounts an attack successfully on B_1 in time U. Then $p_{B_1} \leq Ug(exp(-1/3(b2^{(km/2)-3}/Tg - 1)^2 Tg2^{-km/2})$.*

Proof: By assumption, an adversary can submit at most Tg short solutions in an interval of time Tg. Let X_i be a random variable equal to 1 if the ith short solution submitted by the adversary is correct, and 0 otherwise. Let $X = \sum_{i=1}^{T} gX_i$. By Lemma 1, the probability that a short solution is correct is $\leq 2^{-km/2}$, hence $E[X] = Tg2^{-km/2}$. The attack on buffer memory B_1 is deemed successful if $X \geq b/8$. Thus the probability of a successful attack on B_1 is at most $\mathrm{pr}[X - E[X] \geq b/8 - E[X]] = \mathrm{pr}[X - E[X] \geq (b/(8E[X]-1))E[X]] = \mathrm{pr}[X - E[X] \geq (b2^{km/2}/8Tg - 1)E[X]]$. By Lemma 3 (inequality (1)), then $p_{B_1} \leq exp(-1/3(b2^{(km/2)-3}/Tg - 1)^2 Tg2^{-km/2})$. Since puzzle guess successes are independent of one another, the result follows by a union bound. □

The proof of Lemma 4 relies on the fact that an adversary is unlikely to be able to guess solutions correctly. Lemma 5 treats the case of long solutions. Lemma 5 essentially expresses the fact that an adversary working on many long solutions is likely to take about the expected amount of time to do so.

Lemma 5 *Let us denote by p_{B_2} the probability that the adversary mounts an attack successfully on B_2 in time U. Then $p_{B_2} \leq Ug(exp(-(7bm/64)(1 - Tg/(7bm^{2k-5}))^2))$.*

Proof: In order to attack buffer memory B_2 successfully, the adversary must submit at least $7/8b$ correct long solutions. By Observation 2, this means that the adversary must submit at least $7bm/16$ solved sub-puzzles. By Lemma 2, an optimal adversary will solve these sub-puzzles in strict, sequential order. Recall that the time to solve a sub-puzzle is a uniform random variable uniform on the interval of integers $[1, 2^k]$. Let X_i be a random variable denoting the time required to solve the ith sub-puzzle divided by 2^k. Hence X_i is uniform over [0,1]. Let $X = \sum_i^{7bm/16} X_i$. Thus the time required for the adversary to solve the sub-puzzles is $2^k X$ steps, and $E[X] = 7bm/32$. In order for the adversary's attack to be successful, therefore, it must be the case that $2^k X \leq Tg$. Now, by the Hoeffding inequality given in Lemma 3 (inequality (2)), $\mathrm{pr}[2^k X \leq Tg] = \mathrm{pr}[X \leq Tg2^{-k}] = \mathrm{pr}[X - E[X] \leq -(1 - Tg/(2^k E[X]))E[X]]$ is bounded above by $exp(-1/2(1 - Tg/(2^k E[X]))^2 E[X])) = exp(-(7bm/64)(1 - Tg/(7bm2^{k-5}))^2)$.

Suppose that the adversary has been successful in overloading B_2 in time step $t-1$. Since the adversary submits at most one puzzle per time step, at most one puzzle can expire in any given time step, and hence only one of the buffer slots in B_2 will expire in time step t. Hence, the probability that the adversary is successful in overloading B_2 again in time step t is equal to the probability that it solves a new sub-puzzle in time step t, which is at least 2^{-k}. This is more than the *a priori* probability that the adversary successfully overloads B_2 in time step t. In other words, the probability of success in time step t conditional on success in $t-1$ is greater than the *a priori* probability. It follows that the probability that the adversary overloads B_2 in time step t conditional on failure to overload B_2 in time step $t-1$ must be less than the *a priori* probability. Therefore the result follows by a union bound. □

We can now bound the success probability of the adversary by combining Lemmas 4 and 5. This yields a somewhat complicated expression in the parameters b, k, m, T, g, and U, and implicitly on the ratio of the computing power of the adversary to that of clients. In particular, it is easy to see that a union bound yields the following.

Corollary 2 *Let p be the probability that an adversary mounts a successful attack against the client puzzle protocol in U time steps. Then $p \leq Ug[exp(-1/3(b2^{(km/2)-3}/Tg - 1)^2 Tg2^{km/2}) + exp(-(7bm/64)(1 - Tg/(7bm2^{k-5}))^2)]$.* □

In the interest of producing a simpler, if less general main theorem, we make the following three assumptions on the parameters in the expression of Corollary 2. These assumptions describe a conventional attack model and usage parameters.

1. The adversary has a total computing power of no more than 10^8 MIPS. A time step is defined to be the time for the adversary to execute 400 instructions (i.e., one MD4 computation). It follows that

the adversary executes no more than 2×10^{11} time steps / sec, i.e., that $g \leq 2 \times 10^{11}$

2. The adversary devotes no more than a year to mounting a connection depletion attack. (This combined with the previous assumption yields $Ug < 2^{63}$.)

3. Puzzles are constructed such that $m \geq 8$.

Finally, we use the average time required to solve a puzzle in order to parameterize the number b of buffer slots in B devoted to defending against adversarial attack. Since the expected time to solve a sub-puzzle is 2^{k-1} steps, and a puzzle contains m sub-puzzles, the number of buffer slots an adversary can force the server to allocate "on average" in time T is $Tg/(m2^{k-1})$. We let $c = b/(Tg/m2^{k-1}) = bm2^{k-1}/Tg$. The variable c thus represents the ratio of the chosen buffer size to the buffer size we would expect an adversary to attack successfully on average.

Theorem 1 *Assume an adversary with at most 10^8 MIPS of computing power that mounts a connection depletion attack for no more than a year. Assume further that puzzles are constructed such that $m \geq 8$. Then if $c \geq 7/2$ and $b \geq 1100$, the probability that the adversary is able to mount a successful attack is less than 2^{-100}.*

Proof: Given the conditions $m \geq 8$, $c \geq 7/2$, and $b \geq 1100$, straightforward application of algebra to Lemma 4 shows that $p_{B_1} \leq Uge^{-115}$ (in fact, p_{B_1} is substantially less than Uge^{-115}). Observe that appropriate substitution of c into the exponent in Lemma 5 yields $p_{B_2} \leq Ug(exp(-(7bm/64)(1-16/7c)^2))$. Therefore, when $c \geq 7/2$ and $b \geq 1100$, it is easily seen that $p_{B_2} \leq Uge^{-115}$. By Corollary 2, we can bound the probability that the adversary is able to mount a successful attack by $p \leq 2Uge^{-115}$. By assumption $Ug < 2^{63}$. The theorem follows. □

Remark Theorem 1 requires that $c \geq 7/2$ for the client puzzle protocol to be successful. In other words, the buffer B must be 7/2 times larger than we would expect to defend against an adversary with average success in solving client puzzles. By means of a more fine-grained analysis, it is possible to show that Theorem 1 holds for substantially smaller c. In fact, it holds for c slightly more than 1. Such detailed analysis, though, would result in a much longer proof.

Roughly speaking, there are two factors that enlarge the size of c required by Theorem 1. The first arises from Bounding Assumption 1: we assume that submitting any long solution to a puzzle (rather than a correct long solution) is sufficient to cause a buffer allocation in the server. This assumption enlarges c by about a factor of 2. The second factor arises from Bounding Assumption 3. Our division of B into buffers B_1 and B_2 in the proportion 1 : 7 enlarges c by a factor of about 8/7. (Hence, we would expect the exponent in Lemma 5, for instance, really to be close to $(bm/4(1-1/c))$.) The minimum buffer size b in Theorem 1 is subject to a similar enlargement due to our bounding assumptions. Rather than making direct use of Theorem 1, we recommend that designers of real-world client puzzle protocols make use of the following heuristic.

Heuristic 1 *Assume an adversary with at most 10^8 MIPS of computing power that mounts a connection depletion attack for no more than a year. Assume further that puzzles are constructed such that $m \geq 8$. Then if $c \geq 4/3$ and $b \geq 1000$, the probability that the adversary is able to mount a successful attack is less than 2^{-100}.* □