Efficient Memory Bound Puzzles Using Pattern Databases

Sujata Doshi, Fabian Monrose, and Aviel D. Rubin

Johns Hopkins University, Computer Science Department, MD, USA
{sdoshi, fabian, rubin}@cs.jhu.edu

Abstract. CPU bound client puzzles have been suggested as a defense mechanism against connection depletion attacks. However, the wide disparity in CPU speeds prevents such puzzles from being globally deployed. Recently, Abadi *et. al.* [1] and Dwork *et. al.* [2] addressed this limitation by showing that memory access times vary much less than CPU speeds, and hence offer a viable alternative. In this paper, we further investigate the applicability of memory bound puzzles from a new perspective and propose constructions based on heuristic search methods. Our constructions are derived from a more algorithmic foundation, and as a result, allow us to easily tune parameters that impact puzzle creation and verification costs. Moreover, unlike prior approaches, we address client-side cost and present an extension that allows memory constrained clients (e.g., PDAs) to implement our construction in a secure fashion.

1 Introduction

The Internet provides users a plethora of services, but at the same time, it is vulnerable to several attacks. Denial of Service (DoS) attacks, for example, represent a potentially crippling attack vector by which a user or organization is deprived of legitimate services, and may be forced to temporarily cease operation. While many approaches have been suggested as countermeasures to DoS attacks, one of the more promising avenues for defending against such attacks is based on the notion of client puzzles [3, 4, 5, 6].

Juels *et. al.* present one of the first practical solutions that employs CPU puzzles to defend against connection depletion attacks. [3]. That approach attempts to overcome the limitations of SYN-cookies [7] and random dropping of connections [8] by instead issuing puzzles constructed from time-sensitive parameters, secret information held by the server, and additional client request information.

To date, the design of client puzzles are bound by either CPU or Memory constraints of the client. Memory bound puzzles, however, overcome a notable obstacle in the widespread adoption of client puzzles, namely the large disparity in client CPU speeds. Recently, Abadi *et. al.* [1] proposed the first memory-bound puzzle construction aimed at addressing this disparity, and provide a solution based on performing a depth first search through a large array.

Unfortunately, while that approach and subsequent work [2] has indeed validated the conjecture regarding the low disparity in memory access times, we find

J. Zhou, M. Yung, and F. Bao (Eds.): ACNS 2006, LNCS 3989, pp. 98–113, 2006.

that prior work in this area lacks a thorough algorithmic foundation. Specifically, the constructions presented to date involve accessing random locations in a large array, but unlike some CPU-bound instances, these accesses are not semantically associated to solving any known hard problem. Furthermore, the memory-bound puzzle constructions presented thus far incur high creation and verification costs which themselves can lead to a form of DoS attack. While it may be argued that by appropriately adjusting the parameters of these constructions the task remains memory rather than CPU bound, a rigorous empirical evaluation has yet to be presented.

In this paper we propose a new memory bound puzzle construction based on heuristic search using pattern databases. One of the primary advantages of such an approach is that there already exists an equivalence class of problems (such as the Sliding Tile [9] and the Rubik cube problems [10]) that have been solved efficiently using memory based heuristics [11, 12], and that can be used as building blocks in our constructions. Furthermore, this class of problems enhances the flexibility in controlling the hardness of the client puzzle.[1]

In what follows, we present constructions based on the Sliding Tile problem, but note that it can be easily replaced with an equivalent problem. Additionally, the algorithmic nature of our constructions allows for simple and efficient extensions. Specifically, we consider the case of constrained clients (e.g, PDAs) that may not have sufficient memory to implement our constructions, and propose an enhancement which reduces the memory overhead at the client while still maintaining the security of the scheme.

2 Preliminaries

Our primary goal is to explore memory bound puzzles and appropriate constructions that meet the definition below. For the most part, the properties enlisted have been introduced elsewhere [3, 1, 2, 13], but we restate them here for completeness. We also introduce a new *Relaxed State* property for client puzzles.

Definition 1. ***Memory Bound Client Puzzles*** *are computable cryptographic problems which provide the following properties:*

- ***Stateless:*** *The server can verify the puzzle solution without maintaining state.*
- ***Time-Dependent:*** *The client is allowed limited time range in which the puzzle must be solved.*
- ***Inexpensive Server-Side Cost:*** *Creation and verification of the puzzle is inexpensive for the server.*
- ***Controlled Hardness:*** *The server can control the hardness of the puzzle it sends to the client.*

[1] For instance, the branching factor of the problem controls the number of paths that need to be explored in order to reach a solution. Hence, as technology evolves the building blocks of our constructions can be replaced with ones having a higher branching factor.

- ***Hardware Independent:*** *The puzzle should not be hardware dependent — ensuring that the puzzle can be widely deployed.*
- ***Hardness of Pre-computation:*** *It is computationally hard for the client to pre-compute the puzzle solution. This ensures that while the puzzle can be reused, its solution is not reusable.*
- ***Random Memory Access:*** *A memory bound function should access random memory locations in such a way that the cache memory becomes ineffective.*
- ***Slower CPU bound solution variants:*** *A client puzzle, can be solved by a memory bound or a CPU bound method. However the Memory Bound algorithm should converge faster than the corresponding CPU bound variant.*
- ***Relaxed State:*** *The server is allowed to maintain a limited amount of state for puzzle creation and verification. This property is applicable where additional storage is not a primary concern.*

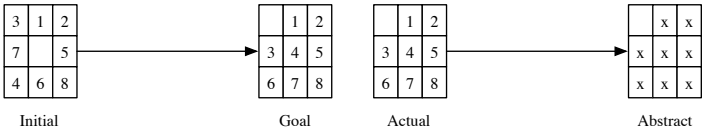
3 Related Work

CPU Bound: Cryptographic puzzles were first introduced by Merkle [14] in the context of key agreement protocols where the derived session key is the solution to the puzzle. Juels *et. al.* [3] further extended the idea of puzzles in an attempt to provide a countermeasure to connection depletion attacks. Essentially, the client is forced to perform multiple hash reversals to correctly solve the puzzle. While [3] addresses server-side issues, little attention is given to client-side overhead.

Another approach to building CPU bound puzzles is the use of Hashcash [15]. HashCash was originally proposed as a countermeasure to email spam, and hence requires non-interactive cost-functions. The drawback, however, of a non-interactive approach is that an attacker can pre-compute all the tokens (solutions) for a given day and temporarily overload the system on that day.

Dean *et. al.* [4] show the applicability of CPU bound puzzles in protecting SSL against denial of service attacks and Wang *et. al.* [16] introduce the notion of congestion puzzles to defend against DDoS attacks on the IP layer. Wang and Reiter [5] address the issue of setting puzzle difficulty in the presence of an adversary with unknown computing power. They present a mechanism of puzzle auctions where each client bids for resources by tuning the difficulty of the puzzles it solves. More recently, Waters *et. al.* [6] point out that the puzzle distribution itself can be subject to attack, and present a defense mechanism which outsources the puzzle via a robust external service called a bastion. Both puzzle auctions and puzzle outsourcing can be adapted to use both CPU and Memory Bound Puzzles.

Memory Bound: Dwork *et. al.* consider memory bound constructions for fighting against spam mail [2]. The basic idea is to accompany email with proof of effort, in order to reduce the motivation for sending unsolicited email. Here, puzzle construction involves traversing a static table T of 2^{22} random 32-bit integers, and the sender is forced to perform a random walk of l steps through this table.



(a) Sliding Tile Problem (b) Mapping from actual to abstract space

Fig. 1. The Sliding Tile Problem and its Abstract Mapping

The walk computes a one-way value R and success is defined when R contains a number of 0's in the least significant bit positions. The recipient accepts the proof (a hash on R and a path identification number i) if i lies under a specific threshold and the hash is correct. While Dwork *et. al.* show that the memory bound running times vary much less compared to the CPU bound variants, a drawback is the high verification cost on the server side. Rosenthal [17] further points out that with Dwork's solution, the sender of an email could have performed less work than that stated in the accompanying proof. To mitigate this scenario a modification was proposed that instead requires the sender to explore an entire range of paths rather than stopping at the first index.

Abadi *et. al.* [1] also propose memory bound constructions which involve accessing random locations in a very large array. There, the server applies a function $F(\cdot)$ k times to a random number, x_0, to obtain x_k. The server then sends x_k and the checksum over the path, $x_0 \cdots x_k$, and a keyed hash $H(K, x_0)$ (where K is a secret key of the server) to the client. Note that the hash is used for verification of the solution sent by the client. The client builds a table of the inverse function $F^{-1}(\cdot)$ and performs random accesses through this table to arrive at x_0. Unfortunately, the construction imposes constant work on the server for puzzle creation which is undesirable.

4 Memory Based Heuristic Search

In this paper we consider heuristic search for the Sliding Tile problem proposed by Sam Lyod [9]. Figure 1(a) illustrates the basic 3x3 Sliding Tile Problem. Here, eight numbers are arranged in a 3 x 3 grid of tiles where one tile is kept blank. The idea is to find a set of moves from the set $\{Left, Right, Up, Down\}$ which transforms the initial configuration to the goal configuration. A widely known, and very efficient, CPU bound method of solving such a problem is to use the A^* algorithm [18, 19, 20] along with the Manhattan Distance heuristic. In this case, a heuristic function $h(s)$ computes an *estimate* of the distance from state s to a goal state. A more efficient way of solving such a problem is to use a memory based heuristic, instead of the Manhattan Distance heuristic, whereby one precomputes the *exact* distance from a state s to the abstract goal state and stores that in a lookup table indexed by s. This lookup table is called a *pattern database* or a *heuristic table*.

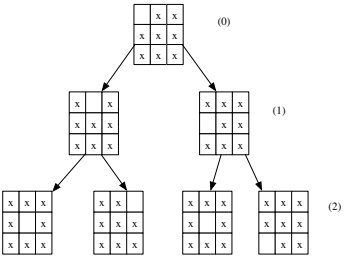


Fig. 2. Pattern Database Creation. The numbers near the configurations depict the heuristic values.

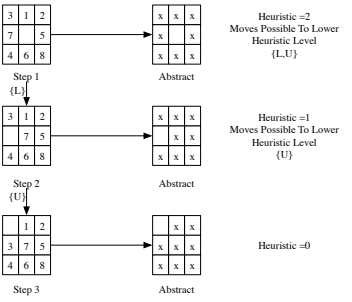


Fig. 3. Solving a Sliding Tile problem with a coarse abstraction

Pattern databases were introduced by Culberson and Schaeffer [21] to find optimal solutions to the 4 x 4 Sliding Tile problem, and have been instrumental in solving large problems efficiently [11]. The primary motivation behind using pattern databases is that they enable search time to be reduced by using more memory [12, 22]. When creating a pattern database, the goal configuration is first mapped to an abstract goal state (as in Figure 1(b)) and then the heuristic values are computed by performing a breadth first search backwards from the abstract goal (as in Figure 2).

5 Memory Bound Constructions

In what follows we describe two constructions—a naïve algorithm that does not meet all the properties of Definition 1, and then extend that to achieve a better construction. Section 5.1 presents the initialization steps and some notation common to both constructions.[2]

5.1 Initialization

The client and server have an agreed upon goal state(s). The client initially precomputes the pattern database corresponding to the goal. For example, Figure 1(b) shows the coarse mapping from the actual state space to the abstract space for the 3 x 3 Sliding tile problem; such a mapping yields a database consisting of 9 heuristic values corresponding to the 9 unique locations of the blank tile.

With this abstraction the database can be used to solve sub-versions of the original problem. However, notice that while the goal state might be reached in the abstract space, the goal might not be reached in the actual state space. Consequently, one will also have to explore search paths in the actual state space

[2] Note that in our protocols we do not focus on adapting the puzzle hardness in accordance with the changing memory size of the client. However, such a mechanism can be incorporated along the lines of the auction protocol provided by Wang and Reiter [5].

without using the database, hence, the client needs to also use an exhaustive CPU bound search to completely solve the problem. Figure 3 illustrates the process of solving the puzzle in Figure 1(a) given a database with this abstraction. Notice that Steps 1 and 2 are memory bound and lead to the goal in the abstract space. However, the client still has to perform additional moves from Step 3 to the actual Goal state. This implies that the given abstraction leads to a partially memory-bound search. Note, however, that a one-to-one mapping between the actual and abstract goal yields a larger pattern database which stores the exact heuristic values and that the corresponding search is completely memory bound. We use such a mapping in the various constructions introduced. Also note that precomputing this database is computationally expensive and hence it must be created offline.

Additionally we assume there exists a publicly accessible random oracle which can be queried to obtain a checksum value C. (In our case the oracle is implemented using a cryptographic hash function). Furthermore the server has access to a pseudorandom function $\mathcal{F}_K(\cdot)$ (such as HMAC-SHA1 [23]) where K is a secret key known only by the server.

5.2 Naïve Construction

The client and server have an agreed upon goal state G. The client precomputes the pattern database corresponding to the goal G. The protocol steps are:

- Puzzle Creation: The server applies d moves at random to G, from the set { *Left, Right, Up, Down* }, to arrive at the configuration P. Let M_i denote the opposite of the i^{th} move on the puzzle where i takes values in $[1, d]$. Note that the parameter d controls the puzzle difficulty. The server computes a checksum C over $(M_d \dots M_1)$.[3] The server also computes a verification value $V = \mathcal{F}_K(T, M_1, \dots M_d)$ where T is the time stamp associated with the client visit. The server sends P, C, V, and T to the client.
- Puzzle Solving: The client uses the pattern database and performs a guided search from P until he reaches the goal G and the checksum over the moves performed from P to G matches C. A guided search essentially involves following paths which lead closer to the goal. The client returns T, V and the d moves $\{M'_1 \dots M'_d\}$ to the server.
- Puzzle Verification: The server verifies that the d moves sent by the client are correct using the verification value $V \stackrel{?}{=} \mathcal{F}_K(T, M'_1 \dots M'_d)$.

Experimental Analysis. We implemented the above construction using the 2 x 4 Sliding Tile problem and evaluated it on machine M6 in Table 1. We chose a 2 x 4 configuration instead of a larger configuration (e.g. 3 x 3), to prevent a pattern database for one goal state from occupying too much main memory. This consideration becomes important in Section 5.3 where the client needs to store a pattern database for a large pool of goal states.

[3] The checksum is computed in the opposite direction over the moves as the client solves the puzzle from P towards G.

Table 1. Machine Specifications

Label	Processor	CPU (GHz)	Cache (KB)	Memory (MB)
M1	Pentium 2	0.4	512	128
M2	PowerMac G4	1.33	256 L2 2048 L3	1024
M3	Pentium 4	1.6	512	256
M4	PowerPC G4	1.67	512	1024
M5	PowerMac G5	2	512	3072
M6	Pentium 4	3.2	1024	1024

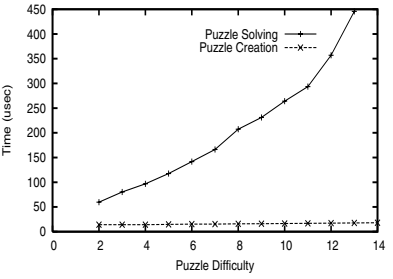


Fig. 4. Client and Server Costs

Figure 4 compares the client versus server cost for varying puzzle difficulty d. It can be seen that the work ratio between the client and server is substantial—for instance, at $d = 12$ moves, the server takes merely $17.3\mu sec$ to create the puzzle while solving the puzzle takes approximately $356.8\mu sec$. Additionally, the server does not maintain any database to verify the puzzle solution and so this construction meets the *Statelessness* property of Definition 1.

To determine if the memory bound approach is more effective than CPU bound methods for *solving* the puzzle, we compare our naïve construction against the best known (to our knowledge) CPU-bound method for solving the Sliding Tile problem — namely, the A^* algorithm[18, 19] with the Manhattan Distance Heuristic. Let $P(x)$ denote the fraction of nodes with heuristic value $\leq x$. If b denotes the branching factor of the problem and d denotes the solution depth (i.e puzzle difficulty) then the average case time complexity of A^* is $\frac{1+\sum_{i=0}^{d} b^i P(d-i)}{2}$ [24, 25]. Note, however, that the regular A^* algorithm does not incorporate the checksum into the search algorithm and so once an optimal path is found the result is returned. On the other hand, our setting requires that the client returns the path for which the checksum matches. In this way, the naive construction forces the client to search through non-optimal paths as well.

Figure 5 compares the time complexity for various search methods in terms of node expansion at a given solution depth. Note that node expansion is a valid metric for complexity because it inherently affects the search time. The results in Figure 5 show that the time complexity of the Naive Construction is higher than that of A^* algorithm with Pattern Database heuristic, indicating that the checksum forces the client to search non-optimal paths, thus confirming our previous argument. Note also that the performance of the Naïve Construction tends to follow the plot of A^* with Manhattan Distance. These results indicate that given an algorithm that incorporates the checksum into A^* with Manhattan Distance we can safely claim that the time complexity for such an algorithm would be more inline with that of the Brute Force approach. As such, we argue that brute force search is indeed a reasonable baseline for comparing our memory-bound approach.

Figure 6 compares the search component of the naïve construction to the brute force depth-first search approach which explores all paths until it reaches the goal, and the checksum matches. The results clearly indicates that the naïve

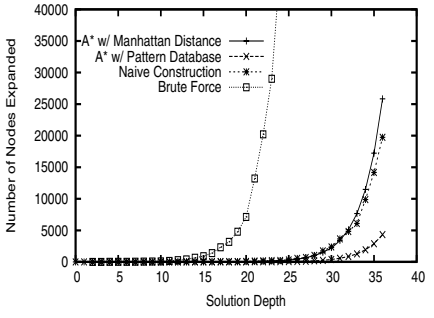


Fig. 5. Time Complexity

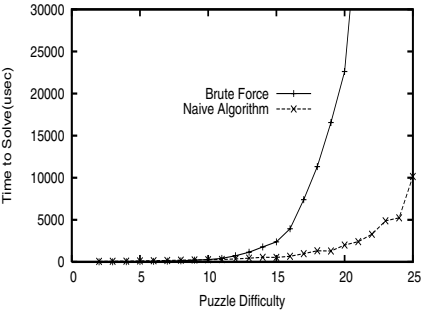


Fig. 6. Naïve vs Brute Force on M6

approach achieves better performance than the brute force algorithm and shows that the construction achieves the *Slower CPU bound solution variants* property of Definition 1. Unfortunately, this naïve construction suffers from a number of significant limitations, most notably that it fails to meet the following criteria:

- **Hardness of Pre-computation:** If the client is presented with an initial configuration P' whose moves to the goal state is a superset of the moves of a previously solved configuration P, then the client can re-use his old solution. While this issue cannot be completely resolved, the probability of re-using old solutions can be reduced by increasing the pool of initial configurations available to the server.
- **Random Memory Access:** The accesses in the pattern database are not random. More specifically, if we consider the number of unique puzzle configurations at a given heuristic level for the 2 x 4 Sliding Tile problem, then the maximum number of configurations between two consecutive heuristic levels is only 2000. This corresponds to at most 100 KB of memory and so the consecutive moves made by the client will not be cache misses. Moreover, the total number of configurations for the 2 x 4 Sliding Tile problem is just over 20,000 which results in a table of roughly 1 MB—which can easily be cached. Hence, this naïve solution is not memory bound.

In what follows we present a variant that overcomes the limitations of the naïve construction.

5.3 A Construction Using Multiple Goals

Again, assume that the client and server have an agreed upon pool of goal states $\{G_0, \ldots G_n\}$. The client precomputes the pattern database corresponding to all of these goal states and stores it in one table. The database is indexed by the tuple $(G_i, P), i \in \{0 \ldots n\}$ where P denotes the current configuration. The indexed location contains the heuristic value — the distance from the current configuration P to the goal configuration G_i. We define puzzle difficulty based on two parameters, namely the *horizontal puzzle difficulty* which denotes the number of Sliding Tile problems that have to be solved simultaneously, and the

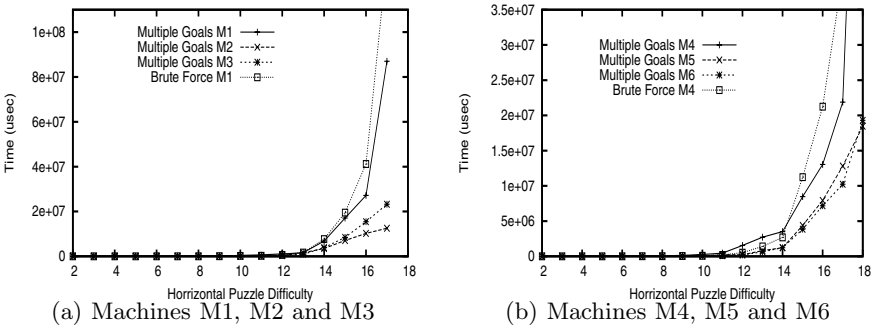
vertical puzzle difficulty which denotes the number of moves required to reach the goal state for a given initial configuration. The protocol steps are:

- PUZZLE CREATION: The server chooses f goal states at random from $\{G_0, \ldots G_n\}$. Let the set $\mathcal{G}$ contain these f goal states. The server then applies d moves at random to each goal $G_k \in \mathcal{G}$, from the set { *Left, Right, Up, Down* }, to arrive at the f initial configurations P_k. Note that the parameter d controls the vertical puzzle difficulty and the parameter f controls the horizontal puzzle difficulty. The server also computes checksums over the moves as follows. Let $M_j^k, 1 \leq j < d$ denote the opposite of the j^{th} move on the goal G_k. For difficulty level $j, 1 \leq j < d$, the checksum C_j is taken over $M_j^k, \forall G_k \in \mathcal{G}$.
  The server also computes a verification value $V = \mathcal{F}_K(T, d \text{ moves over } f \text{ configurations })$ where T is the time stamp associated with the client visit. The server sends the goal configurations chosen $G_k \in \mathcal{G}, |\mathcal{G}| = f$, corresponding initial configurations P_k, checksums $C_j, 1 \leq j \leq d$, V, and T to the client.
- PUZZLE SOLVING: The client uses the pattern database and performs a guided search from each P_k. The client solves all the initial configurations simultaneously. This implies that the client first infers the right set of moves $M_j^k, \forall G_k \in \mathcal{G}$ for a given difficulty level $j, d \geq j \geq 1$ such that the checksum over those moves matches C_j. Then he proceeds to do the same for the next level $j-1$. This procedure is followed until the client reaches the goal configurations $G_k \in \mathcal{G}$. The client returns T, V and the d moves over these f initial configurations to the server.
- PUZZLE VERIFICATION: The server verifies that the d moves for all the f Sliding Tile problems sent by the client are correct using the verification value $V \stackrel{?}{=} \mathcal{F}_K(T, d \text{ moves over } f \text{ configurations })$.

If b is the brute force branching factor of the problem, then in the worst case, the time complexity of our multiple goals construction is $O((b-c)^f d)$ where c is a constant that depends on the number of paths pruned by the pattern database heuristic at a given horizontal level.

Experimental Analysis. We now evaluate this construction using a pool of 100 configurations for the 2 x 4 Sliding Tile problem on the machines given in Table 1. The pattern database for 100 configurations took approximately 30 minutes to build on M6 indicating that the database must be created offline. Note, however, that this is a one time cost. The database occupies around 169MB of the main memory and cannot be cached given that typical cache sizes are less than 8MB. Figures 7(a) and 7(b) compares the cost for solving a puzzle with a brute force search and a memory bound search against varying horizontal difficulty f. There, the vertical difficulty was set at $d = 20$ which yields a larger pool of 1194 sliding tile puzzles (per goal configuration) to choose from.

Observe that even though the worst case time complexity of brute force is $O(b^f d)$, up to a horizontal difficulty of $f \leq 15$ brute force search is more effective



(a) Machines M1, M2 and M3 (b) Machines M4, M5 and M6

Fig. 7. Multiple Goals vs Brute Force

on all the machines.[4] However, beyond $f = 15$ our memory bound approach is considerably faster suggesting that for $d = 20$ the horizontal difficulty should be set above 15. Moreover, our results show that the time for solving a puzzle is on the order of seconds for $f \in [15, 18]$, indicating that our construction performs as well as or better than prior work. For example, in the solution presented by Waters *et. al.* [6] a client may need to wait for roughly 20 minutes before she can gain access to server resources. In addition, our solution is considerably better than that of [1] in terms of puzzle solving times, indicating that our approach reduces end user wait time compared to Abadi's approach.

Comparison with HashCash: For completeness, we also compared our results with the CPU bound algorithm, HashCash. Table 2 indicates the time to solve a puzzle with parameters, $f = 16$, $d = 20$ and a pattern database for 100 2 x 4 goal configurations. We compare our results with the time to mint 100, 20 bit hash cash tokens—more than 20 bit tokens take considerably longer to mint. Our results show that with the memory bound approach, the disparity in puzzle solving times across machines is much less when compared to HashCash. Specifically, the maximum ratio of the time to solve a CPU bound puzzle (HashCash) across machines is 9.17, but only 5.64 in the memory bound case. Furthermore, the puzzle solving times are much lower in the memory bound case—our slowest machine (M1) takes 291.88 seconds to mint a HashCash token versus only 33.91 seconds to solve the memory bound puzzle—indicating that our approach may be even better suited for global deployment than HashCash.

Comparison to the Naïve Approach: Unlike the naïve approach, this alternative does meet the *Random Memory Access* property of Definition 1. This is achieved by choosing f goals at *random* from the available pool of goals. This ensures that the pattern databases corresponding to each of these f goals would not be located at contiguous regions in memory. The client is thus forced to access these random locations when solving the Sliding Tile problems simultaneously.

[4] On M2, M3 and M5 the bound is at $f = 15$. On machines M4 and M6 brute force is more effective up to $f = 14$ and on M1 the bound is at $f = 13$.

Table 2. Memory Bound vs Hash Cash

Machine	Memory Bound (seconds)	Ratio	Hash Cash (seconds)	Ratio
M1	33.91	**5.64**	291.88	**9.17**
M2	14.75	2.45	71.01	2.23
M3	14.2	2.36	152.65	4.8
M4	17.44	2.9	37.9	1.19
M5	8.93	1.48	78.38	2.46
M6	6.01	1	31.8	1

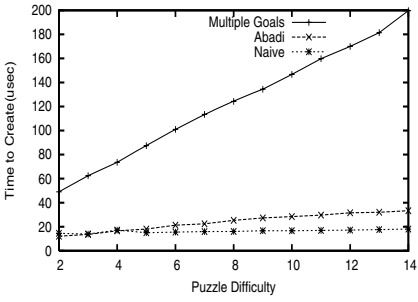


Fig. 8. Puzzle Creation Costs

As it stands, the approach does not meet the property of *In-expensive Server Side Cost* of Definition 1. Specifically, the puzzle creation cost in this approach remains high. Figure 8 compares the cost of creating a puzzle to that of Abadi *et. al.* [1]. While the naïve algorithm outperforms Abadi's approach, puzzle creation in the multiple goals approach is 5 times as slow. This high work factor can in itself lead to a DoS attack. In what follows, we show how server side puzzle creation cost can be reduced considerably. We note that in doing so we forgo the *Stateless* property in Definition 1 but achieve the *Relaxed state* property suggesting that our construction is still of practical value.

5.4 Reducing Server Side Cost

In puzzle creation and verification the server side work involves performing d moves on f goal configurations, computing d checksums (hashes), a MAC to compute the verification value V, and a final MAC for puzzle verification. This overhead is unfortunately much higher than [1] which involves only d moves, 1 checksum, a hash for the verification value, and a final hash for puzzle verification. We can address this limitation by having the server create p puzzles offline, and storing these puzzles in a table. Each location of the table simply contains:

1. **Puzzle to be sent to the client.** The goal configurations chosen $G_k \in \mathcal{G}, |\mathcal{G}| = f$, corresponding initial configurations P_k, checksums $C_j, 1 \leq j \leq d$
2. **Solution to Puzzle.** The d moves for the f goal configurations $G_k \in \mathcal{G}, |\mathcal{G}| = f$

On a client visit, the server computes the time stamp T associated with the visit and generates a random number R. The server computes an index into the table I choosing $\log(p)$ bits of $\mathcal{F}_K(T, R)$ deterministically. Recall that $\mathcal{F}_K(\cdot)$ is a pseudo random function and K is the server's secret key. The server sends to the client, T, R and the puzzle at index I. The client returns the solution to the puzzle with T and R and the server simply recomputes the index I and verifies that the solution sent by the client matches the solution at index I. In doing so, we reduce the online server work for puzzle creation and verification to computing two MAC's per client connection. This server work is a slight improvement over Abadi *et. al* which adds additional online work on the server

Table 3. Server storage per puzzle

State Table Items	Storage (bytes)
16 Goal Configurations	32
16 Initial Configurations	32
20 Checksums	400
Puzzle Solution (20 moves for 16 configurations)	320
Total Storage per Puzzle	784

Table 4. Memory Read Time

Machine	Read Time (μsec)
M1	0.27
M2	0.18
M3	0.20
M4	0.27
M5	0.26
M6	0.15

for computing the d moves and the checksum over the path. Furthermore, this approach outperforms that of Dwork *et. al.* [2] which adds additional overhead on the server side during puzzle verification.

In the following discussion we show how an upper bound on the parameter p is obtained, depending on the expected server load.

Setting the state parameters: Similarly to [3], we assume that the server issues puzzles to defend against TCP SYN-flooding attacks. Let τ denote the total time for the client server protocol, including the time for which the TCP buffer slot is reserved. Let the server buffer contain p additional slots for legitimate TCP connections when under attack. Note that the client induces at least fd memory accesses when solving a puzzle. Hence, to solve p puzzles an adversary will take at least pfd time steps. To mount a successful attack the adversary must solve p puzzles in τ seconds. If m denotes the number of memory accesses that an adversary can perform per second, then to prevent a flooding attack p should be set to $\frac{\tau m}{fd}$. This is an upper bound on p, because fd is the minimum number of memory accesses required to solve a given puzzle.

Assuming that a client server connection takes around $\tau = 150$ seconds [3] and that the average read time is $0.2\mu sec$ (see Table 4), then this allows for $m = 5000000$ accesses per second. Hence, for $d = 20$ and $f = 16$ a maximum of $p = 2,500,000$ puzzles need to be created offline. Additionally, each puzzle requires the server to store 784 bytes of information (see Table 3). Under these parameters the resulting state table is at most 1.96 GB, which can be easily stored in the main memory of a storage server. This confirms that our construction is practical.

5.5 Improving Client Side Cost

Earlier we noted that the pattern database for heuristic values corresponding to 100 goal configurations is approximately 169 MB in size. For some devices, however, 169 MB is prohibitively large. As such, it is desirable to have a method by which the pool of configurations available stays the same, but the size of the pattern database is reduced. Specifically, it would be ideal if a given pattern database could be used for multiple goal configurations. Intuitively, we can do so as follows: assume that a client has a pattern database for goal configuration G. Our task is to adapt this pattern database for goal configuration G'. One

way of achieving this is by providing the client a hint in the form of the relative distance r (either positive or negative) between G and G'. The client augments the heuristic values stored in the table with r when performing the guided search to the goal G'. In this case, the protocol steps are now as follows:

- Puzzle Creation: The client and server have an agreed upon pool of goal states $\{G_0 \ldots G_n\}$. The client maintains pattern database corresponding to these goal states. The server picks f goal states at random from the pool (say $\mathcal{G}$ contains these f goals) and performs a set of d moves from the f goals in $\mathcal{G}$ to arrive at the corresponding initial configurations. This operation is the same as presented in Section 5.3. The server also performs a set of r moves from the randomly chosen goals to the actual goal states $\mathcal{G}'$. The checksum is computed over all levels starting from the actual goal states up to the initial configurations. Along with this puzzle the server sends a hint, r, which is the distance between the actual goal states in $\mathcal{G}'$ and the database goal states in $\mathcal{G}$ and the verification value V as before.
- Puzzle Solving: The client performs a guided search as before, but also augments the heuristic values with this relative distance, r, when deciding which path should be followed.
- Puzzle Verification: The server uses the verification value V to verify that the $d + r$ moves for all the f Sliding Tile problems are correct.

Adding relative distance thus allows the client to use the same pattern database for multiple goal configurations, and still meets all the properties of Definition 1. Additionally, this enhancement provides more flexibility in controlling vertical difficulty of a puzzle. Furthermore, the simplicity of this extension is an added benifit of our algorithmic approach.

We argue that considering both the client and the server side improvements the multiple goals construction offers a viable memory bound puzzle construction.

6 Security Analysis

In this section we informally justify the claims that our constructions meet the (security) properties outlined in Section 2. Note that the justifications assume a computationally bounded adversary $\mathcal{A}$.

Claim 1. *The Sliding Tile problem is more efficiently solved with a memory bound approach (i.e., A^* with pattern database heuristic) compared to the best known CPU bound approaches (to date).*

Korf *et.al* [11] showed that memory based heuristics for this class of problem provide a significant reduction in search time at the cost of increasing the available memory. Specifically, if n denotes the number of states in the problem space and m the amount of memory used for storing the heuristic values, then the running time t of A^* is governed by the expression $t \approx n/m$. This analysis was later revisited by Holte *et. al.* [22] who subsequently showed a linear relation between

$log(t)$ and $log(m)$. Specifically, as m increases, the number of states explored in the heuristic search diminishes, which inherently reduces the running time. These results [11, 22] show that memory bound heuristics are indeed the most efficient method to date to reduce search time, and so we argue that Claim 1 is satisfied. To address the security of the underlying approach we first restate the properties of a pseudo random function [26].

Definition 2. *A cryptographically secure pseudorandom function $\mathcal{F}_K(\cdot)$ is an efficient algorithm that when given an l-bit key, K, maps n-bit argument x to an m-bit string such that it is infeasible to distinguish $\mathcal{F}_K(x)$ for random K from a truly random function.*

Claim 2. *The multiple goals scheme is secure against an adversary, $\mathcal{A}$, in the random oracle model as long as Claim 1 holds and the verification value, V, is the output of a pseudorandom function.*

Following from Claim 1, and assuming that the pattern database heuristic is computed using a one-to-one mapping between the abstract and actual state space (see Section 5.1), then $\mathcal{A}$ can not solve the puzzle faster using a CPU bound approach. Furthermore, even though $\mathcal{A}$ can perform multiple queries to the random oracle, it is computationally hard to determine information about the underlying moves from C_i. In addition since V is computed using a pseudo random function $\mathcal{F}_K(\cdot)$, it is difficult for $\mathcal{A}$ to determine the moves, considering that K is a secret random key of the server.

Claim 3. *A parallelizable solver can not solve the puzzle more efficiently than a brute force approach when puzzle difficulty is set appropriately.*

To see why that is the case, assume that $\mathcal{A}$ uses multiple processes to simultaneously solve the multiple goal configurations. Note that in order to arrive at a correct solution, the moves obtained by each process must collectively match the checksum. In other words, given the set of moves obtained by each process, $\mathcal{A}$ needs to determine the correct permutation of these moves that will match the given checksum. However, the process of determining the correct set is essentially a brute force search, which we showed to be ineffective for $d = 20$ and $f > 15$.

7 Conclusion

In this paper we introduce the first heuristic search based memory bound puzzle, and present several constructions accompanied by rigorous experimental analysis. Our constructions address the issues of non-reusable solutions, random memory access, and easily parametrized client and server-side tuning. Additionally, we present several improvements to our multiple goals construction that limit server and client side overhead. From the client's perspective, we also address a major concern regarding limited memory on constrained clients such as PDAs, and present an enhancement that allows the client to use the same

pattern database for multiple goals—without violating the general properties of client puzzles. Our client puzzle protocol is interactive and hence is applicable to defend against DoS attacks such as TCP SYN flooding. Exploring methods to extend our construction to defend against spam and DDoS attacks remains a possible area of future work.

Acknowledgements

We thank Patrick McDaniel and Seny Kamara for their invaluable feedback on this research. We also thank the anonymous reviewers for their insightful comments on this paper. This work is funded by the NSF grant CNS-0524252.

References

1. Abadi, M., Burrows, M., Manasse, M., Wobber, T.: Moderately hard, memory-bound functions. In: Proceedings of Network and Distributed Systems Security Symposium, San Diego, California, USA. (February 2003) 107–121
2. Dwork, C., Goldberg, A., Naor, M.: On memory-bound functions for fighting spam. In: Proceedings of the 23^{rd} Annual International Cryptology Conference. (2003) 426–444
3. Juels, A., Brainard, J.: Client puzzles: A cryptographic countermeasure against connection depletion attacks. In: Proceedings of Networks and Distributed Security Systems. (February 1999) 151–165
4. Dean, D., Stubblefield, A.: Using client puzzles to protect TLS. In: Proceedings of the 10^{th} USENIX Security Symposium. (August 2001) 1–8
5. Wang, X., Reiter, M.K.: Defending against Denial-of-Service attacks with puzzle auctions. In: Proceedings of the IEEE Symposium on Security and Privacy, IEEE Computer Society (2003) 78–92
6. Waters, B., Juels, A., Halderman, J.A., Felten, E.W.: New client puzzle outsourcing techniques for DoS resistance. In: Proceedings of the 11^{th} ACM conference on Computer and Communications Security. (2004) 246–256
7. Bernstein, D.J.: SYN cookies (1996) `http://cr.yp.to/syncookies.html`.
8. Floyd, S., Jacobson, V.: Random early detection gateways for congestion avoidance. IEEE/ACM Transactions on Networking **1(4)** (1993) 397–413
9. Loyd, S.: Mathematical Puzzles of Sam Loyd. Dover (1959) Selected and Edited by Martin Gardner.
10. Singmaster, D.: Notes on Rubik's Magic Cube. Enslow Pub Inc. (1981)
11. Korf, R.: Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In: Proceedings of the 14^{th} National Conference on Artificial Intelligence and 9^{th} Innovative Applications of Artificial Intelligence Conference, Rhode Island, AAAI Press/MIT Press (July 1997) 700–705
12. Hern'advolgyi, I.T., Holte, R.C.: Experiments with automatically created memory-based heuristics. In: Proceedings of the 4^{th} International Symposium on Abstraction, Reformulation, and Approximation, London, UK, Springer-Verlag (2000) 281–290
13. Bocan, V.: Threshold puzzles: The evolution of DoS-resistant authentication. Periodica Politechnica, Transactions on Automatic Control and Computer Science **49(63)** (2004)

14. Merkle, R.C.: Secure communications over insecure channels. Communications of ACM **21(4)** (April 1978) 294–299
15. Back, A.: Hash cash - A Denial of Service Counter-Measure. Technical report (2002) `http://www.hashcash.org/`.
16. Wang, X., Reiter, M.K.: Mitigating bandwidth-exhaustion attacks using congestion puzzles. In: Proceedings of the 11^{th} ACM conference on Computer and Communications Security, New York, NY, USA, ACM Press (2004) 257–267
17. Rosenthal, D.S.H.: On the cost distribution of a memory bound function. Computing Research Repository **cs.CR/0311005** (2003)
18. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics **4(2)** (1968) 100–107
19. Hart, P.E., Nilsson, N.J., Raphael, B.: Correction to a formal basis for the heuristic determination of minimum cost paths. ACM SIGART Bulletin (37) (1972) 28–29
20. Parberry, I.: A real-time algorithm for the $(n^2 - 1)$-puzzle. Information Processing Letters **56**(1) (1995) 23–28
21. Culberson, J.C., Schaeffer, J.: Searching with pattern databases. In: Advances in Artificial Intelligence, 11^{th} Biennial Conference of the Canadian Society for Computational Studies of Intelligence, Springer (1996) 402–416
22. Holte, R.C., Hern'advolgyi, I.T.: A space-time tradeoff for memory-based heuristics. In: Proceedings of the 16^{th} national conference on Artificial Intelligence and the 11^{th} Innovative Applications of Artificial Intelligence conference, Menlo Park, CA, USA, American Association for Artificial Intelligence (1999) 704–709
23. FIPS: The Keyed-Hash Message Authentication Code (HMAC). (2002) `http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf`.
24. Korf, R.E.: Recent progress in the design and analysis of admissible heuristic functions. In: Proceedings of the 17^{th} National Conference on Artificial Intelligence and 12^{th} Conference on Innovative Applications of Artificial Intelligence, AAAI Press / The MIT Press (2000) 1165–1170
25. Korf, R.E., Reid, M.: Complexity analysis admissible heuristic search. In: Proceedings of the 15^{th} national/10^{th} conference on Artificial Intelligence/Innovative Applications of Artificial intelligence, Menlo Park, CA, USA, American Association for Artificial Intelligence (1998) 305–310
26. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. J. ACM **33**(4) (1986) 792–807