Moderately Hard, Memory-bound Functions

MARTIN ABADI
University of California at Santa Cruz
and
MIKE BURROWS
Google
and
MARK MANASSE and TED WOBBER
Microsoft Research, Silicon Valley

A resource may be abused if its users incur little or no cost. For example, e-mail abuse is rampant because sending an e-mail has negligible cost for the sender. It has been suggested that such abuse may be discouraged by introducing an artificial cost in the form of a moderately expensive computation. Thus, the sender of an e-mail might be required to pay by computing for a few seconds before the e-mail is accepted. Unfortunately, because of sharp disparities across computer systems, this approach may be ineffective against malicious users with high-end systems, prohibitively slow for legitimate users with low-end systems, or both. Starting from this observation, we research moderately hard functions that most recent systems will evaluate at about the same speed. For this purpose, we rely on memory-bound computations. We describe and analyze a family of moderately hard, memory-bound functions, and we explain how to use them for protecting against abuses.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*security and protection*; H.4.3 [**Information Systems Applications**]: Communications Applications—*electronic mail*

General Terms: Algorithms, Security

Additional Key Words and Phrases: Spam

1. INTRODUCTION

With the increase in the number of e-mail users and the proliferation of junk e-mail (spam), several techniques for discouraging or filtering spam have been proposed (e.g., [Ahn et al. 2003; Cranor and LaMacchia 1998]). In particular, Dwork and Naor suggested in their seminal paper that a way to discourage spam is to force senders of e-mail to pay by performing a moderately expensive computation [Dwork and Naor 1992]. More recently, Back rediscovered this idea and implemented it in the HashCash system [Back 1997] (see

A preliminary version of this work has been presented in a conference [Abadi et al. 2003]. Most of Martín Abadi's and Mike Burrows's work was done at Microsoft Research, Silicon Valley (where the techniques described in this paper were invented), with Microsoft's support. Martín Abadi's work is also partly supported by the National Science Foundation under Grant CCR-0208800.

also [CAMRAM 2002]).

Their basic scheme goes as follows. Suppose that sender S is sending an e-mail M to recipient R. We assume that M includes normal mail headers, in particular the time and R's address. If R has previously agreed to receive e-mail from S, then M is sent in the normal way. Otherwise, they proceed:

—S computes some moderately-hard function $G(M)$ and sends $(M, G(M))$ to R.

—R verifies that what it receives from S is of the form $(M, G(M))$. If so, R accepts M. If not, R bounces M, possibly indicating in the bounce message where S can obtain software for computing $G()$.

The function $G()$ is chosen so that the verification by R is fast, taking a millisecond, say, and so that the computation by S is fairly slow, taking at least several seconds. Therefore, S could be (somewhat) discouraged from sending M. For a spammer that wishes to send many millions of messages, the cost of computing $G()$ repeatedly can become prohibitive.

Such schemes, with refinements and extensions, have a variety of interesting applications. For example, moderately expensive computations also play a role in secure classification [Gabber et al. 1998], another scheme for curbing spam. Beyond combating spam, requiring moderately expensive computations can help in protecting against other abuses. For example, Web indexes could require a computation each time a user tries to add a URL to the index, thereby limiting additions; a server could require a computation each time a client tries to establish a connection, thereby countering connection-depletion attacks [Juels and Brainard 1999]. A paper by Jakobsson and Juels discusses several other applications and develops a formalization of the concept of proof of work [Jakobsson and Juels 1999].

In some cases, it is preferable that S apply a moderately hard function to a challenge provided by R (rather than to a particular message or request):

—S contacts R, requesting permission to use some service.

—R returns a fresh challenge x to S.

—S computes $G(x)$ and returns it to R.

—R verifies that what it receives is a correct response to x. If so, R allows S to use the service.

This variant enables S to compute $G(x)$ well before actually using the service in question.

In previous work in this area, the emphasis is on CPU-intensive computations. In particular, Dwork and Naor suggest CPU-intensive candidates for the function $G()$ such as breaking the Fiat-Shamir signature scheme with a low security parameter. Back's HashCash scheme relies on the brute-force search for partial collisions in a hash function.

The starting point for the present paper is a simple, new observation about a problematic feature of such moderately hard computations. Fast CPUs run much faster than slow CPUs—consider a 2.5GHz PC versus a 33MHz Palm PDA. Moreover, in addition to high clock rates, higher-end computer systems also have sophisticated pipelines and other advantageous features. If a computation takes a few seconds on a new PC, it may take a minute on an old PC, and several minutes on a PDA. That seems unfortunate for users of old PCs, and probably unacceptable for users of PDAs. While it is conceivable that service providers may (for a fee) perform computations on behalf of users of low-end machines, such arrangements are not ideal. These arrangements would conflict with free e-mail, and may be unstable: service providers could save money and trouble by making contracts to pass e-mail between themselves without actually performing the computations. So, moderately hard computations may be most appropriate when performed by clients. Therefore, we believe that the disparity in client CPU speed constitutes one of the significant obstacles to widespread adoption of any scheme based on a CPU-bound moderately hard function.

In this paper, we are concerned with finding moderately hard functions that most computer systems will evaluate at about the same speed. We envision that high-end systems might evaluate these functions somewhat faster than low-end systems, perhaps even 2–10 times faster (but not 10–100 faster, as CPU disparities might imply). Moreover, the best achievable price-performance should not be significantly better than that of a typical legitimate client. We believe that these ratios are egalitarian enough for the intended applications: the functions should be effective in discouraging abuses and should not introduce a prohibitive delay on legitimate interactions, across a wide range of systems.

Our approach is to rely on memory-bound functions.[1] A memory-bound function is one whose computation time is dominated by the time spent accessing memory. The ratios of memory latencies of machines built in the last five years is typically no greater than two, and almost always less than four. (Memory throughput tends to be less uniform, so we focus on latency.) A memory-bound function should access locations in a large region of memory in an unpredictable way, in such a way that caches are ineffectual. This strategy can work only if the largest caches are significantly smaller than the smallest memories across the machines of interest. Unfortunately, one can now buy machines with 8MB caches, and some PDAs have only 8MB of memory or less, so perhaps there is little or no room to manœuvre. On the other hand, at the time of this writing, machines with 8MB caches are still expensive rarities, while PDAs with 64MB of memory are fairly common. So we proceed by restricting our attention to machines with at least 32MB of available memory. In light of technology commonalities, PDA memories may grow as fast as caches over the next few years.

The next section, section 2, further describes our approach; it explores a particular class of memory-bound computations related to inverting functions. Section 3 develops this approach into a complete method. Section 4 presents some variants of the method. Section 5 then investigates specific instances of the method. Section 6 gives experimental results. Section 7 concludes, mentioning some other related work and some open questions.

In our presentation, we emphasize the application of memory-bound functions to discouraging spam. However, memory-bound functions are immediately applicable in protecting against other abuses (for example, against abusive additions of URLs to Web indexes and against connection-depletion attacks). In particular, a future release of Microsoft's Passport system may use our functions as one of the mechanisms for controlling account creation. Memory-bound functions are also applicable for strengthening passwords. We explain this application, which is less straightforward, in section 4. Other possible applications include establishing shared secrets over insecure channels and the timed release of information, using memory-bound variants of Merkle puzzles [Merkle 1978] and of time-lock puzzles [May 1993; Rivest et al. 1996], respectively. We discuss these also in section 4.

[1] It appears that this approach has not previously been explored. However, in January 2003, after performing much of this work, we discovered a brief mention of memory-bound functions in the camram mailing list [Astley 2001].

2. MEMORY-BOUND COMPUTATIONS: INITIAL IDEAS

Our approach is to force the sender S to access an unpredictable sequence of locations in a large array. The size of this array is chosen to be significantly larger than the largest cache available; at present, the size of the array should be at least 16MB.

One possibility is to prescribe a computation on some large data structure, for example a large graph, that would force the desired memory accesses. Unfortunately, with this strategy, the definition of the function may itself become rather large and hard to communicate, and checking S's answer may be costly. Nevertheless, this strategy might be viable.

An alternative, which we adopt, is to prescribe a computation that could be done with very little memory but which is immensely helped by memory accesses. More specifically, let $F()$ be a function whose domain and range are integers in $0..(2^n - 1)$, where 2^n is the number of entries in the array. This function will vary from time to time; for example, R may choose a fresh $F()$ periodically. Suppose that $F()$'s inverse $F^{-1}()$ cannot be evaluated in less time than a memory access. If we ask S to compute $F^{-1}()$ many times, then it becomes worthwhile for S to build a table for $F^{-1}()$ and to rely on the table thereafter.

The table can be computed by 2^n applications of $F()$. Building the table also requires memory accesses, for storing the table entries. However, these memory accesses can benefit from batching, and their cost (like that of applying $F()$) is not necessarily uniform across machines. Therefore, the cost of building the table should not be dominant in S's work in responding to R's challenge. Rather, the dominant cost should be that of performing many table lookups.

In order to develop these initial ideas, we first describe a naive embodiment and list some of its problems (section 2.1). Then we make an interesting but imperfect improvement (section 2.2). We design and study a complete method later in this paper.

2.1 A naive embodiment

A naive embodiment of our ideas consists in letting R challenge S with k values $x_0, \ldots,$ x_{k-1}, and requiring S to respond with their immediate pre-images, that is, with values $y_0,$ $\ldots, y_{k-1}$ such that $F(y_0) = x_0, \ldots, F(y_{k-1}) = x_{k-1}$.

This naive scheme is flawed, in at least four respects:

(1) The size of the challenge is $n \times k$. While n will not be very large, because 2^n will be smaller than the memory size, k will need to be quite large so as to determine a sufficiently difficult problem. The resulting size of the challenge could be on the order of megabytes. Therefore, the challenge would be hard to transmit to S.

(2) If the values $x_0, \ldots, x_{k-1}$ are all presented at once, a brute-force search can attempt to find pre-images for all of them at once, by computing $F()$ forward. This search will require at most 2^n computations of $F()$—a large number, but probably not large enough. If k is small enough, $x_0, \ldots, x_{k-1}$ will be cached rather than stored in memory, so this brute-force search will be CPU-bound and it will be faster than the expected memory-bound computation. If k is large, so $x_0, \ldots, x_{k-1}$ are stored in memory, the brute-force search will require memory accesses, but these can be organized in such a way that their cost is not uniform across machines.

On the other hand, if R presents $x_0, \ldots, x_{k-1}$ sequentially, waiting for S's response to x_i before giving x_{i+1}, the naive approach requires a prohibitively large number (k) of rounds of communication.

(3) If R must present the challenge to S, then S is unable to prepare a message to be sent to R without first contacting R. While this interaction may be acceptable in some circumstances, we would like to have the option of avoiding it. One technique for avoiding it, which we exploit in a system currently under development [Abadi et al. 2003], consists in letting a trusted third party present the challenge to S; but, in some settings, a suitable trusted third party may not be easy to find.

(4) The ratio of the work done at S and R is the ratio in time between a memory access and a computation of $F()$. This ratio is unlikely to be more than 10, and cannot be more than 100 or so with present machines. (Here we ignore the cost of building a table at S, since it should be dominated by the cost of the later lookups in the table, as indicated above.) A higher ratio is desirable in order to limit R's work.

2.2 An improvement: chains

Chaining the applications of $F()$ helps in addressing shortcomings 1 and 2 of the naive scheme. (We return to shortcomings 3 and 4 in later sections.) The chaining may go as follows:

—R picks a value x_0.

—R computes x_k by letting, for all $i \in 0..(k-1)$,

$$x_{i+1} = F(x_i)$$

—R gives x_k to S and challenges S to find x_0.

The hope is that, as long as 2^n and k are large enough, the fastest approach for S would be to perform k accesses into a table to evaluate $F^{-1}()$ as many times. S should perform these accesses in sequence, not because of interaction with R but because each access depends on the previous one. The function $F()$ should be such that the sequence of accesses has poor locality and is hard to predict, so S should not benefit from caches. Finally, the size of the challenge x_k (n bits) is smaller than in the naive scheme.

This straightforward use of chains is however unsatisfactory. In particular, if the sequence of values produced by successive invocations of $F()$ contains cycles smaller than 2^n, then S might be able to use those cycles as shortcuts. On the other hand, if $F()$ is a permutation with a single cycle of length 2^n, then S may find x_0 from x_k with at most $k + 2^{n+1}$ forward computations of $F()$ and hardly using memory:

```
$x$ := an arbitrary value;
$y$ := $F^k(x)$;
while $y \neq x_k$ {
    $x$ := $F(x)$;
    $y$ := $F(y)$;
}
return $x$;
```

In order to defeat this CPU-based solution and to eliminate cycles, we change the recurrence to depend on the step number. In what follows, we use:

$$x_{i+1} = F(x_i) \text{ xor } i$$

The xor with the step number permutes the values at each step; we have also experimented with other permutations, for example xoring a function of the step number rather than the step number itself.

Even after this correction, the design of a scheme based on chains requires further elaboration. In particular, when the function $F()$ is not itself a permutation, there may be many valid responses to the challenge x_k: there may be many x'_0 such that the recurrence $x'_{i+1} = F(x'_i)\ \text{xor}\ i$ yields $x'_k = x_k$. We should specify which of these x'_0 are acceptable responses.

This difficulty can be addressed by generalizing from chains to trees, as we do next. The generalization also allows us to avoid the other shortcomings of the naive scheme of section 2.1.

3. A COMPLETE METHOD: TREES

Building on the ideas of the previous section, we design and study a method that relies on trees.

3.1 The method

In trying to address the shortcomings of chains, we work with functions that are not permutations, so we need to specify which are the acceptable responses to a challenge x_k. At least two approaches are viable:

—One approach is to accept not only x_0 but all x'_0 such that the recurrence $x'_{i+1} = F(x'_i)\ \text{xor}\ i$ yields $x'_k = x_k$. It is still useful to construct x_k from x_0, rather than completely at random, in order to ensure that at least one acceptable response exists. This approach obviously adds to the cost of verifying a response, since remembering x_0 does not suffice for the verification: R needs to follow a chain from x'_0 to x_k.

—Another approach, which we prefer, is to accept only x_0, forcing S to explore a tree of pre-images rather than a chain of pre-images. The tree has root x_k and depth k. The nodes of the tree are (immediate or iterated) pre-images of x_k. One of the leaves at depth k is x_0.

This presents a further problem, namely that S does not know which of the many possible leaves at depth k is R's chosen one. S could perhaps send all of these leaves to R, but this would add considerable communication cost. (The number of these leaves can be fairly large.)

A solution is for R to provide S with a cheap checksum of the path from x_k to x_0. This checksum should be such that S can tell when it has found x_0, yet the checksum should not allow S to prune the space of possibilities in advance of a search. The checksum need not be cryptographically strong.

In summary, the resulting method is as follows:

—Let k and n be two integers, and let $F()$ be a function whose domain and range are integers in $0..(2^n - 1)$. We suppose that $F()$'s inverse $F^{-1}()$ cannot be evaluated in less time than a memory access. We assume that k, n, and $F()$ are known to both R and S, possibly because R has chosen them and communicated them to S.

—R picks an integer x_0 in $0..(2^n - 1)$ and computes, for $i \in 0..(k - 1)$:

$$x_{i+1} = F(x_i)\ \text{xor}\ i$$

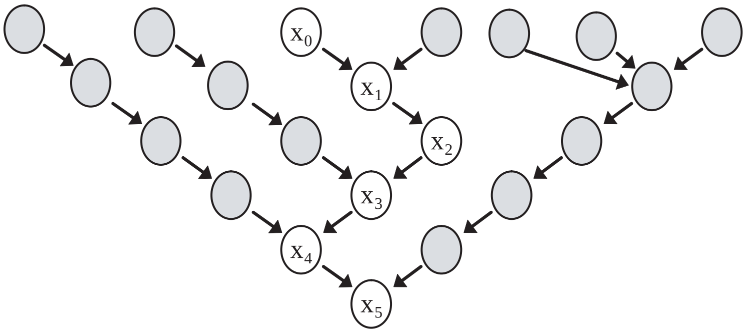


Fig. 1. An example tree

and a checksum of the sequence $x_0, \ldots, x_k$. R sends x_k and this checksum to S.

—With this information, S should find x_0 and return it to R.

—When R receives a response from S, it simply checks that it is x_0.

Figure 1 depicts an example tree with $k = 5$; each arrow corresponds to an application of $F()$ and an xor with the appropriate step number.

We expect S to proceed as follows in order to find x_0:

—Construct a table for $F^{-1}()$ by applying $F()$ to all integers in $0..(2^n - 1)$.

—Build sequences $y_k, \ldots, y_0$ starting with $y_k = x_k$ and such that

$$y_i \in F^{-1}(y_{i+1} \text{ xor } i)$$

(so that $y_{i+1} = F(y_i) \text{ xor } i$).

—Given such a sequence, return y_0 if the checksum matches.

S may build the sequences $y_k, \ldots, y_0$ depth-first (hoping to find a match early, much before building all sequences); or S may build them breadth-first (trying to hide some of the memory latency). In either case, S should perform many accesses to the table for $F^{-1}()$.

Of course, S may instead adopt alternative, CPU-intensive algorithms. However, when $F()$, n, and k are chosen appropriately, we believe that S's task is memory-bound. In other words, those CPU-intensive algorithms will be slower than a memory-bound solution. We do not unfortunately have a formal proof of this conjecture, and it may be quite difficult to obtain one (see section 7 for a brief discussion of other approaches). Below, we give calculations that support this conjecture focusing on particular CPU-intensive algorithms.

3.2 Trees and work

The ratio of the work done at S and R is greatly improved when we force S to explore a tree as explained above. Thus, the use of trees also addresses problem 4 of section 2.1. In this section we analyze that work ratio. We also calculate the expected performance of S using alternative, CPU-intensive algorithms. We obtain some constraints on n, k, and other parameters. Figure 2 shows some of the notation used in this section.

$F()$	a function with domain and range $0..(2^n - 1)$
x_i	the value obtained after i applications of $F()$
n	the size of each value
k	the number of times R applies $F()$ to generate a problem
f	the cost of applying $F()$
f_0	the cost of applying $F()$ on a slow machine
f_1	the cost of applying $F()$ on a fast machine
c	a factor for work performed by S
p	the number of problems over which S amortizes a table for $F^{-1}()$
r	the cost of a memory read
w	the cost of a memory write

Fig. 2. Notation

A quadratic factor. In order to characterize the work ratio, it is helpful to be more specific on the basic function $F()$. An interesting possibility, which we discuss further in section 5.1, is to let $F()$ be a random function. (Here, and in the rest of this paper, we say that $F()$ is a random function if and only if $F(x)$ is uniformly distributed over $0..(2^n - 1)$, for each x, and independent of all $F(y)$ for $y \neq x$.)

When $F()$ is random and $k < 2^n$, the size of the tree explored by S is quadratic in k, so S is forced to perform far more work than R even if it takes as long to compute $F()$ as $F^{-1}()$. Basically, the size of the tree is approximately $k^2/2$, and S needs to explore half of the tree on average (with depth-first search), so S needs to evaluate $F^{-1}()$ roughly $k^2/4$ times on average. In contrast, R applies $F()$ only k times.

More precisely, we have made the following observation. Suppose that the function $F()$ on $0..(2^n - 1)$ is random and $k << 2^n$. Let x_0 be a random value and let x_k be defined by the recurrence:

$$x_{i+1} = F(x_i) \text{ xor } i$$

Construct a tree with root x_k and in which, if y is at depth $j < k$ from the root, then z is a child of y if and only if

$$y = F(z) \text{ xor } (k - j - 1)$$

The expected number of leaves of this tree at depth k is approximately $k + 1$. The expected size of this tree is approximately $(k + 1)(k + 2)/2$. These numbers require that the tree in question be constructed from a leaf x_0, rather than from a random root x_k: the expected size of a tree grown from a random x_k is considerably smaller.

We have noticed the quadratic size of trees in experiments, letting $F()$ be various practical (not exactly random) functions. Section 6 discusses these experiments further. A posteriori, we have sketched a proof of the quadratic size, there assuming an independent random function at each tree level. (In short, our proof sketch inductively argues that the expected number n_j of nodes at depth j is almost $j + 1$ as long as j is much smaller than 2^n: by construction, x_{k-j} is a node at depth j, and for $j > 0$, each of the $2^n - 1$ other values maps to one of the n_{j-1} nodes at depth $j - 1$ with probability $n_{j-1}/2^n$, so $n_j = 1 + (2^n - 1) \times n_{j-1}/2^n = 1 + n_{j-1} - (n_{j-1}/2^n)$.) A more sophisticated analysis might be possible using tools from research on random functions, a rich field with many theorems (see for instance [Flajolet and Odlyzko 1990]).

In light of the quadratic size of trees, it is tempting to use very deep trees, so as to increase the work ratio between S and R. There are, however, important limitations on tree depth. At each level in a tree, S may try to invert all the leaves simultaneously, somehow. When there are enough leaves, S may benefit from cache behaviour. Specifically, when several leaves land in the same cache line, the cost of inverting all of them is essentially the cost of just one memory access. (A cache line is the unit of cache replacement, and is typically around 64 bytes.) These issues are particularly clear when k nears the size of the space, 2^n. We must therefore keep k much smaller than 2^n—say, below 2^{n-5}.

Some calculations. Next we derive a few simple formulas that (roughly) characterize the work at R and—using several different algorithms—at S. We obtain some constraints on n, k, and other parameters. We indicate some precise values for parameters in section 5.2.

For simplicity, we assume that R has chosen $F()$ and communicated it to S; sections 3.3 and 5.1 say more on the choice of $F()$. We also rely on the quadratic ratio established above. We assume that k is "small enough" (in particular, so that this ratio applies). Finally, we assume that checksumming is essentially free (partly because we do not require a strong cryptographic checksum). We write f for the cost of one application of $F()$, r for the cost of one memory read (with a cache miss), and w for the cost of one memory write.

—R's cost in making a challenge will essentially be that of k applications of $F()$, that is, $k \times f$.

—S's cost for building a table for $F^{-1}()$ will be that of:
—2^n applications of $F()$;
—2^n insertions into the table.
Naively, this cost appears to be $2^n \times (f + w)$. However, for some functions $F()$, the cost of 2^n applications of $F()$ may be substantially smaller than $2^n \times f$. Similarly, the cost of inserting 2^n entries may be substantially smaller than $2^n \times w$, because the necessary writes can be batched and completed asynchronously by the hardware. On the other hand, if the table structure is similar to that of a hash table, then the insertions will require reads in order to resolve collisions. These reads may make the cost of building the table closer to $2^n \times (f + r)$. In the calculations below, we assume that the cost is $2^n \times (f + w)$ and we often assume that $w = r$.

—S's cost for solving a challenge using a table for $F^{-1}()$ and depth-first search will be approximately that of $k^2/4$ memory accesses without significant help from caches, that is, $(k^2/4) \times r$.

—If S prefers not to use a table for $F^{-1}()$, it may still follow the same search strategy by pretending that it has a table and by inverting $F()$ on the fly (by brute force) whenever necessary. Provided that an inversion of $F()$ requires 2^n applications of $F()$, the cost of this CPU-intensive approach will be $k^2 \times 2^n \times f$. With a little more trouble, a CPU-intensive search may be done only once for each level in the tree of pre-images, with total cost $k \times 2^n \times f$.

—If S prefers not to use a table for $F^{-1}()$, S may also guess x_0 and check its guess by applying $F()$. For each guess, it has to apply $F()$ k times, so the expected cost of this CPU-intensive approach will be that of $2^{n-1} \times k$ applications of $F()$, that is, $k \times 2^{n-1} \times f$.

—Along similar lines, S may apply $F()$ only $\sqrt{k}$ times to each of the values in $0..(2^n - 1)$; because of collisions, roughly $2^{n+1}/\sqrt{k}$ distinct values will remain after this, and S may then apply $F()$ to them $(k - \sqrt{k})$ times (terminating half way through these applications, on average). The expected cost of this more sophisticated (but realistic) CPU-intensive approach will be $(\sqrt{k} \times 2^n + 2^{n+1}/\sqrt{k} \times (k - \sqrt{k})/2) \times f$, that is, $(2 \times \sqrt{k} - 1) \times 2^n \times f$.

—S may be able to find other optimizations of the brute-force, CPU-intensive search for x_0. In particular, in order to minimize applications of $F()$, S may try to notice collisions after each round of applications of $F()$ (rather than only once after $\sqrt{k}$ rounds). Thus, S would apply $F()$ to each of the 2^n values just once, then apply $F()$ only once to each of their images, and so on. S may thus require $c(k) \times 2^n$ applications of $F()$, where $c(k)$ is an affine function of the logarithm of k. Conceivably, this and other optimizations can lead to a cost of $c \times 2^n \times f$, where c is a small integer (say, below 10). Note however that this is a coarse bound on ambitious, speculative ideas, not a measurement of an actual efficient implementation: we do not know how to realize these ideas without substantial overhead.

We arrive at the following constraints:

(1) *Efficient table building:* As indicated in section 2, the cost of building the table for $F^{-1}()$ should not be dominant in the table-based solution. Suppose that S amortizes a table over p problems. (Section 3.3 says more on amortization.) Then we should have

$$p \times (k^2/4) \times r \ \geq \ 2^n \times (f + w)$$

that is,

$$k \ \geq \ 2^{(n/2)+1} \times \sqrt{1/p} \times \sqrt{(f+w)/r}$$

This lower bound can be reduced when, as suggested above, the cost of 2^n applications of $F()$ and 2^n stores is smaller than $2^n \times (f + w)$.

(2) *No fast CPU-intensive search:* We would like the table-based solution to be faster than the CPU-intensive solutions. With the simpler CPU-intensive solutions, this condition means roughly that

$$k \ \leq \ 2^{n+1} \times (f/r)$$

With the more sophisticated CPU-intensive solution described above, however, we should have that

$$k \ \leq \ (2^{n+3} \times (f/r))^{2/3}$$

Finally, fearing that one could eventually implement a CPU-intensive solution with cost $c \times 2^n \times f$, we would want

$$k \ \leq \ 2^{(n/2)+1} \times \sqrt{f/r} \times \sqrt{c}$$

(Here we simply ignore the cost of building a table for $F^{-1}()$, since it will be dominated by other costs.)

(3) *High work ratio:* We would also like that setting a challenge is much cheaper than solving it. In other words, $(k^2/4) \times r$ should be much larger than $k \times f$, so k should be much larger than $4 \times (f/r)$. This constraint is easily satisfied when k is large.

(4) *No fast inversion of $F()$:* Another constraint follows from our requirement that $F^{-1}()$ cannot be evaluated in less time than a memory access. Obviously, $F^{-1}()$ can be evaluated with 2^n applications of $F()$, so we must have that $f \geq r/2^n$, but $r/2^n$ will be tiny. A more sophisticated construction permits evaluating $F^{-1}()$ with a much smaller

number of applications of $F()$. Since Hellman introduced this construction [Hellman 1980], it has been extended and improved in several respects (e.g., [Fiat and Naor 2000; Quisquater et al. 2003; Oechslin 2003]); we base the following discussion on Fiat's and Naor's variant [Fiat and Naor 2000], which is attractive because of its generality and rigorous analysis:

For $j = 1..l$, S would precompute m pairs $(x, h_j^l(x))$ where $h_j(x) = g_j(F(x))$ and each $g_j()$ is an auxiliary function. The integers m and l should be such that $l^2 \times m$ is around 2^n and such that $l \times m$ pairs $(x, h_j^l(x))$ can be cached. Therefore, l will be at least 2; we can force it to be larger (at least 3, perhaps 6) by increasing the size ratio between the smallest memory and the largest cache under consideration. In order to find one immediate pre-image of y, S would apply each function $h_j()$ to y up to l times, hoping to hit some precomputed $h_j^l(x)$, then S would reach an immediate pre-image of y by working forward from the associated x. This process can be repeated to find all immediate pre-images of y, with some probability [Naor 2002]. Making the conservative assumption that the applications of the functions $g_i()$ are free and that there is no other overhead, S may evaluate $F^{-1}()$ in time $l^2 \times f$. If S has a huge cache, then l could conceivably be 2, so S could evaluate $F^{-1}()$ in time $4 \times f$. On the other hand, naively, S may keep half of a table for $F^{-1}()$ in a cache of the same size, and thus S may evaluate $F^{-1}()$ in time $r/2$ on average. Under these assumptions, we should require that $4 \times f \geq r/2$, that is, $f \geq r/8$.

Although these assumptions may appear fairly extreme, we believe that it is safer to keep $f \geq r/8$, and we may have to raise this bound in the future. Fortunately, this bound is not particularly problematic, as we demonstrate below.

(5) *Fast computation of $F()$:* On the other hand, f cannot be very large (or else some of the CPU-intensive solutions can be sped up). If applying $F()$ naively is slower than a memory read, then S may build a table for $F()$. Many of the accesses to the table might be organized in big linear scans and might therefore be relatively cheap. Moreover, part of the table might be cached, even across problems that use the same or related $F()$'s, thus further reducing the effective cost of calculating $F()$. Therefore, we consider $f \leq r$.

In the lower bound on k (constraint 1), the value of f should correspond to a slow machine; in the upper bound (constraint 2) and in the other constraints, to a fast machine. (We assume, pessimistically, that attackers have fast machines; we can also assume that the challenges are set at fast servers.) In order to avoid ambiguities, let us call the values of f on slow and fast machines f_0 and f_1, respectively.

There exists a satisfactory value of k provided that:

$$2^{(n/2)+1} \times \sqrt{\frac{1}{p}} \times \sqrt{\frac{(f_0 + w)}{r}} \;\leq\; 2^{(n/2)+1} \times \sqrt{\frac{f_1}{r}} \times \sqrt{c}$$

In other words, we should have:

$$(1/p) \times ((f_0 + w)/r) \;\leq\; (f_1/r) \times c$$

that is,

$$p \;\geq\; (f_0 + w)/(f_1 \times c)$$

For instance, when $c = 4$, $w = f_1$, and $f_0 = 100 \times f_1$, we require roughly $p \geq 25$. With these values, $r = w$, and $n = 22$ (for a realistic memory size), we may let k be 2^{13}.

The corresponding cost is that of 2^{24} memory accesses for each of p problems. Section 5.2 says more on the setting of parameters and their consequences.

The constraints

$$r/8 \leq f_1 \leq r$$

are easy to satisfy. In particular, as CPU speeds increase, we can modify or replace $F()$ in order to slow it down and to preserve $r/8 \leq f_1$. If slow machines are never upgraded, this change will result in a larger f_0, so it may affect both the setting and the solving of challenges on those machines, though in tolerable ways:

—Because of the quadratic factor in the work ratio, setting challenges will remain efficient even on a fairly slow machine. Moreover, it seems reasonable to assume, as we do above, that setting challenges will normally be done at fast machines such as mail servers.

—The modified function $F()$ may compute the images of a variable number of inputs at the same time, as we describe in section 5.1. In this case, the building of a table for $F^{-1}()$ need not be penalized by the modification: it can be as fast as with the original, faster function.

Even without this technique, we can easily accommodate large disparities between the speeds at which clients may build the table. The example settings in which $f_0 = 100 \times f_1$ show that we can support clients that are much slower than those accepted by most users and current applications.

3.3 Refinements

The description above focuses on defining a single problem with a given function $F()$. However, problems may be grouped, and the function $F()$ should not be fixed. Next we consider these important refinements.

Varying the function $F()$. The function $F()$ should vary from time to time. If the same $F()$ were used for many challenges, then CPU-intensive solutions would become attractive. In particular, the CPU-intensive forward searches of section 3.2 (which are described there as though $F()$ is completely fresh) could solve several problems with the same $F()$ simultaneously. Therefore, the function $F()$ should ideally vary from problem to problem. It may be freshly generated for each problem, at random from some family.

The variation of $F()$ may simply consist in xoring a different quantity for each problem. Thus, given a master function $MF()$ and an integer $j \in 0..(2^n - 1)$, R may define a new function $F()$ simply by:

$$F(x) = MF(x) \text{ xor } j$$

The integer j may be a problem index (a counter) or may be generated at random. In either case, if R and S know the master function $MF()$ in advance, then R needs to transmit only j to S in order to convey $F()$. Moreover, as long as $MF()$ remains fixed, S may use a table for $MF^{-1}()$ instead of a table for each derived function $F^{-1}()$, thus amortizing the cost of building the table for $MF^{-1}()$. The master function itself should change from time to time—we may not trust any one function for long.

Of course, there are many other ways of defining suitable families of functions. We return to this matter in section 5.1.

Using multiple functions requires conventions for describing them, for example so that R can tell S about a new function. If $F()$ is derived from a master function and an integer parameter (as in $F(x) = MF(x) \text{ xor } j$), then the description of $F()$ might be a description of the master function plus the parameter. The description of the master function might simply be a short name, if it is well known, or it might be code or a table for the function. The integer parameter can be omitted when it is clear from context, for instance when it is a counter.

Using several problems as a challenge. R may ask S to solve several problems of the sort described above, so that S has more work to do, without increasing the expected difficulty of each problem. In addition to requiring more work, the use of several problems also gives some valuable protection against variability in problem hardness.

We may be concerned that S could amortize some work across several problems and solve them all in parallel with a CPU-intensive approach. Indeed, some flawed variants of our method allow such undesirable amortizations. Two twists thwart these amortizations:

—As described above, the function $F()$ should vary from problem to problem. All the problems in a group may share a master function. For instance, with functions of the form $F(x) = MF(x) \text{ xor } j$, the problems in a group may all share $MF()$ but each may have a different j.

—Each problem's challenge and function description (except the first) may be presented encrypted under a key derived from the path to the solution of the immediately preceding problem.

3.4 Making problems even harder

In addition to grouping problems, other techniques may contribute to making challenges harder. We briefly sketch and speculate on two such techniques in this section.

Omitting bits from problems. One can often make problems harder by omitting some bits from them. In particular, R could omit some bits of the challenge x_k, of the description of the function $F()$, or both, and S would need to guess or reconstruct the missing bits in finding x_0. For instance, R could present the full x_k and a checksum of the path from x_k to x_0, and R could tell S that $F()$ has a definition of the form

$$F(x) = MF(x) \text{ xor } j$$

where S knows $MF()$ but not the integer j; then S may need to try many possible values of j in order to find x_0.

Omitting bits slows down S's memory-bound search. On the other hand, omitting bits does not always slow down CPU-intensive alternatives. For example, CPU-intensive forward searches are not affected when R omits bits from x_k but not $F()$. Therefore, such variants should be used with caution.

Mixing functions. Another way to make problems harder is to interleave applications of multiple functions $F_0(), \ldots, F_m()$. When R constructs the challenge x_k from x_0, at each step, it may apply any of those functions. Thus, for all $i \in 0..(k-1)$, we have $x_{i+1} = F_j(x_i) \text{ xor } i$ for some $j \in 0..m$. S knows $F_0(), \ldots, F_m()$, but not in which sequence R applies them, or not entirely. For instance, S may know that R always applies $F_0()$ except that every 10 steps R applies either $F_0()$ or $F_1()$. Therefore, S basically has to guess (part of) the sequence of function choices when it tries to find x_0.

This technique seems viable. It helps in thwarting certain CPU-intensive attacks and it may yield an improvement in work ratios, at the cost of some complexity.

4. VARIANTS

The tree-based method can also be adapted to scenarios in which the parties or their interactions are somehow constrained. Next we describe variants of the tree-based method that address such constraints. Some of them have to do with the secure communication of secrets or their timed release, rather than with protection against spam and other abuses.

4.1 Forgetting the challenge

Relying on a standard technique, we can save R from remembering x_0 after it sends it to S. Specifically, R can produce a keyed hash $H(K, x_0)$ of x_0, using a cryptographically strong keyed hash function $H()$ [Menezes et al. 1996] and a key K known only to R, and give $H(K, x_0)$ to S along with the challenge. S should return both x_0 and $H(K, x_0)$, so R can check that S's response is correct by recomputing $H(K, x_0)$ from K and x_0. Further refinements of this method can include a proof of freshness, a description of $F()$, and other information with x_0.

4.2 A non-interactive variant

Going further, we return to problem 3 of section 2.1, that is, we show how to avoid requiring R to interact with S before S can send its message M.

If R (or a trusted third party) cannot present a challenge to S, then the challenge can be defined by the message M, as follows.

—S is required to apply a one-way hash function to M (or selected parts of M that suffice for preventing replays, for example R's address and a message identifier).

—Using the result of that hash as the seed to a cryptographic random number generator, S then generates a function $F()$ and a start position x_0 for its tree search.
(If R or a trusted third party can provide a small, partial challenge to S, then S should use it in the choice of $F()$ and x_0.)

—S computes x_k by evaluating $F()$ k times, with the recurrence:

$$x_{i+1} = F(x_i) \text{ xor } i$$

—S must supply a value x'_0 other than x_0 such that $x'_{i+1} = F(x'_i)$ xor i yields $x'_k = x_k$, and that some other, sufficiently rare property holds.
An example of such a property might be that the checksum of the path from x_k to x'_0 be 0 mod 2^m for a sufficiently large m. When 2^m is smaller than k, it is likely that such an x'_0 exists. When no such x'_0 exists, S can pick a new x_0 and $F()$ and try again.
If R verifies that the x'_0 presented by S has the property, and that S did not discard too many functions, then R can be reasonably certain that S had to search a substantial fraction of the tree rooted at x_k.

We may choose a property that is quite hard to satisfy, so as to increase the work that S has to do in finding a suitable x'_0. Despite S's additional effort, its response can remain small.

Alternatively, should S need to do more work than that represented by solving a single problem, S may solve several problems. The problems may all be independently derived from M (each with its own function $F()$ and its own x_0 and x'_0), or they can be linked together (so the answer x'_0 for one problem may be used in computing the function $F()$ and the start position x_0 for the next problem). In either case, S should supply all the values x'_0.

4.3 Strengthening passwords

Interestingly, some of the same ideas can be used for strengthening passwords. In this application, S and R interact before S does its work, but S need not respond to R.

In outline, a method for strengthening passwords goes as follows [Manber 1996; Abadi et al. 1997]. Suppose that two parties, S and R, initially share a password P (possibly a weak password). In order to supplement P, R picks an n-bit password extension Q, where n is an integer parameter. Then R poses a problem with solution Q to S. The problem should be such that S can solve it, with moderate effort, by using P, but such that Q is hard to find without P. Afterwards, S and R share not only P but also Q. In particular, S may use P and Q without further interaction with R, for instance in order to decrypt files that R has previously encrypted. For password extensions longer than n bits, each n-bit fragment may be communicated separately, with P as base password, or sequentially, with P and previous fragments as base password; the latter choice limits parallel attacks, so it seems preferable.

The previous instances of this method require a CPU-intensive computation from S. Unfortunately, this computation may need to be long in order for P and Q to be secure against attackers with faster CPUs.

Next we describe an alternative instance of the method in which S performs a memory-bound computation instead.

—R derives a function $F()$ from the password P (and possibly a salt and some other, public data), chooses an n-bit password extension Q, and lets x_0 be Q.

—R computes x_k by evaluating $F()$ k times, with the recurrence:

$$x_{i+1} = F(x_i) \text{ xor } i$$

R also finds some x'_0 other than x_0 that also maps to x_k in this way.

—R then gives to S a checksum of the path from x_k to x_0 (but neither x_0 nor x_k), and x'_0.

—Using P, S derives $F()$, builds a table for $F^{-1}()$, uses x'_0 and $F()$ to compute x_k, then uses x_k and the table to find x_0, that is, Q.

An attacker that tries to find Q by guessing possible values of P will have to do a memory-bound computation for each such value. Had $F()$ been independent of P, this property would of course not hold. Had R transmitted x_k rather than x'_0, this property would probably not hold either: an attacker with a wrong guess of P would use a wrong $F()$ in constructing a tree of pre-images for x_k, the resulting tree rooted at x_k would probably be small, so the attacker would probably get stuck and eliminate the wrong guess rather quickly. That is why R should provide x'_0. Although finding x'_0 is a non-trivial burden, R may explore only a fraction of the tree of pre-images of x_k for this purpose. Alternatively, R may be able to guess x'_0 and verify that it maps to x_k; if the tree that contains x_0 has l leaves at depth k, then R will succeed after approximately $2^n/l$ guesses.

An attacker that guesses P incorrectly may detect that this guess is incorrect, with some probability, when it fails to find a path with the expected checksum. This possibility may be undesirable, although the attacker may have other, cheaper ways of detecting that its guess is incorrect. So it is attractive to use only weak checksums, so that paths with the expected checksums will always be found, or to drop checksums entirely as in the following alternative protocol:

—S and R derive a function $F()$ from the password P (and possibly a salt and some other, public data), and both build a table for $F^{-1}()$.

—S and R choose random values x_S and x_R, respectively, exchange them, and let $x_0 = (x_S \text{ xor } x_R)$.

—S and R compute x_k by evaluating $F()$ k times, again with the recurrence:

$$x_{i+1} = F(x_i) \text{ xor } i$$

They then find all x'_0 that map to x_k in this way. The password extension Q is a function of all these x'_0 (for example, a hash of all of them except x_0).

Here, both S and R perform the same (expensive) steps to compute a password extension. Undoubtedly, other protocols of this form are viable.

As usual, the cost of building tables can be amortized over multiple searches. The multiple searches might be unrelated to one another; or they might all be part of the same search for an n-bit password extension (for instance, if some bits are omitted from problems); or each search might serve to find an n-bit fragment of a longer password extension.

4.4 More on communicating and releasing secrets

More generally, there are several settings in which computation is supposed to introduce delays in the release of information. (The schemes for strengthening passwords provide one family of examples; there are others.) In this section we briefly consider two possible applications of memory-bound computations in this area, for establishing shared secrets over insecure channels and for the timed disclosure of information.

Memory-bound Merkle puzzles. In the early days of public-key cryptography, Merkle invented an ingenious technique that allows two parties S and R to establish a shared secret in the presence of eavesdroppers [Merkle 1978]. In outline, Merkle defined the following procedure. S generates some number of puzzles. Solving each of these puzzles should be moderately hard. In addition, each puzzle has a name (or index), and finding this name from the puzzle should be moderately hard. S should know how to map puzzle names to the corresponding solutions. For instance, S may generate the puzzles from their names and solutions, and may keep a table with those names and solutions. Upon receipt of the puzzles, R picks one, solves it, and sends back the corresponding puzzle name—but not the solution—to S. Afterwards, S and R may rely on the solution as a shared secret. An attacker who eavesdrops and who wishes to learn the secret should have to solve many of the puzzles before discovering which one R picked.

Thus, S and R should spend considerably less time setting and solving puzzles than the attacker. In particular, the ratio of work between R and the attacker is determined by the number of puzzles. If this number is sufficiently large (say, 2^{30}, as Merkle suggested) then we may be comfortable ignoring any disparities in computational power between R and the attacker. On the other hand, a large number of puzzles may result in a high cost of generating puzzles for S and in a high communication overhead.

The use of memory-bound computations should reduce the effect of computational disparities, and should therefore enable the use of a smaller number of puzzles. As few as 2^{20} or even 2^{10} puzzles may yield a reasonable advantage for R over the attacker, even if the attacker has more computational power than R, provided the resulting shared secret is not long-lived.

In particular, the memory-bound computations for Merkle puzzles may consist in finding particular leaves in trees defined as in section 3. By now, Merkle's technique appears to be of historical interest only; we therefore do not elaborate on this memory-bound variant (for example, on how to tune its parameters concretely).

Memory-bound time-lock puzzles. Timed-release cryptography is concerned with how to encrypt a message so that it cannot be decrypted by anyone until a pre-determined amount of time has elapsed [May 1993; Rivest et al. 1996]. Its goal is to "send information into the future", for example enabling the sealing of auction bids and of sensitive historical records. In these applications, the pre-determined amount of time may be a few weeks or months, but it may also be many years.

It is usually envisioned that the expected delay would be enforced by requiring CPU-bound computations with care that these computations should not be parallelizable. Of course, CPU power is far from constant over the years, and it may also vary widely across the parties that are expected to solve a particular time-lock puzzle.

It therefore seems attractive to consider the possibility of basing time-lock puzzles on memory-bound computations. In order to realize a memory-bound time-lock puzzle, we may for instance specify memory-bound tree searches, like those of section 3, and let their results serve as one-time pad for the sensitive information in question.

However, we have yet to devise a variant of our problems with the required properties and degrees of hardness—there is a large gap between the few seconds of computation that may suffice for discouraging spam and the years of computation that should delay the release of sensitive historical records.

5. INSTANTIATING THE METHOD

In this section, we describe a concrete instantiation of our method of section 3.1. We discuss the choice of a basic function $F()$. We also discuss settings for other parameters, and their motivations and effects.

5.1 Choosing the function $F()$

We would like a function $F()$ that can be evaluated efficiently, but which nevertheless cannot be inverted in less time than a memory cache miss. These two constraints are not too hard to satisfy; next we explore some particular choices of $F()$ and their features.

Random functions. We would like $F()$ to approximate a random function, in order to defeat caches and to obtain reasonable work ratios. An appealing possibility is to let $F()$ be a random function. In this case, we envision that $F()$ could simply be given by a table (without much attention to the random process that generated the table).

The use of a random function $F()$ gives rise to performance issues. Specifically, evaluating a random function may not always be cheap enough. In general, each computation of $F()$ may require a memory access, just like each computation of $F^{-1}()$. The ratio between the work done at S and R will still be quadratic in k, but without the constant factor that represents the difference between the respective costs of evaluating $F()$ and $F^{-1}()$. Although the tree search performed by S forces S to perform substantially more work than R, we may want to increase this difference by our choice of the function $F()$. On the other hand, we may also increase this difference by raising k: the upper bound on k in section 3.2 is greater when $F()$ is slower.

The use of a random function $F()$ also gives rise to a storage problem. In general, R will need to have a table for $F()$. This requirement may be inconvenient.

Finally, the use of a random function $F()$ gives rise to a communication problem. If the choice of function should change from time to time, then it is helpful for the function to have a succinct description, so that it can be communicated efficiently. True random functions do not in general have such succinct descriptions. Therefore, we may not generate and transmit a brand new, random $F()$ for each challenge. Instead, we may derive a challenge-specific function $F()$ from a random master function $MF()$, with a definition like

$$F(x) = MF(x) \text{ xor } j$$

(as discussed in section 3.3). In this case, assuming that $MF()$ is known in advance, only j needs to be transmitted.

Approximations. More generally, we may define:

$$F(x) = G(t, x)$$

where $G()$ is a suitable master function (random, or random enough), and t is a parameter. For such functions, describing $F()$ amounts to giving the corresponding t if $G()$ is known in advance. In addition, evaluating $G()$ and therefore $F()$ may well be cheap. These functions $F()$ may share many of the advantages of true random functions. However, they complicate analysis.

We have investigated several candidate functions $F()$ of this form. Some are based on functions $G()$ from the cryptography literature: one-way hash functions such as MD5 and SHA, or variants of fast encryption algorithms such as TEA [Menezes et al. 1996]. For instance, given a value x, we may apply SHA to a key and to x, then extract $F(x)$ from the result.

Since our main intended applications do not require much cryptographic strength, we have also investigated some faster functions $F()$ of the same form. One is as follows:

—Assuming that n is even, let t_0 and t_1 be two tables of $2^{n/2}$ random 32-bit numbers. Together, t_0 and t_1 play the role of t above.

—Let the bitstring representing x be formed from the concatenation of the bitstrings a_0 and a_1, each of length $n/2$ bits.

—Then let $F(x)$ be the middle bits of the 64-bit product of the two 32-bit numbers indexed by a_0 and a_1 in tables t_0 and t_1:

$$F(x) = \text{middle-bits } (t_0[a_0] * t_1[a_1])$$

This function is depicted in Figure 3.

The tables t_0 and t_1 have only $2^{n/2}$ entries, so they will fit in the cache on most machines. Thus, the evaluation of $F()$ will take only a few cycles. In fact, this function is so fast that it conflicts with the condition $f \geq r/8$ of section 3.2; it is easy to define slower variants of this function that satisfy the condition.

In an early version of our work, the two tables t_0 and t_1 were identical. That saves space for R, but enables S to use a smaller table for $F^{-1}()$ because $F(a_0 \mid a_1) = F(a_1 \mid a_0)$. (Here, we write $a_0 \mid a_1$ for the concatenation of a_0 and a_1.) So letting t_0 and t_1 be identical is not attractive. In that early version of our work, we also used tables of 32-bit primes, rather than tables of arbitrary 32-bit numbers. Primes seem to yield a somewhat better

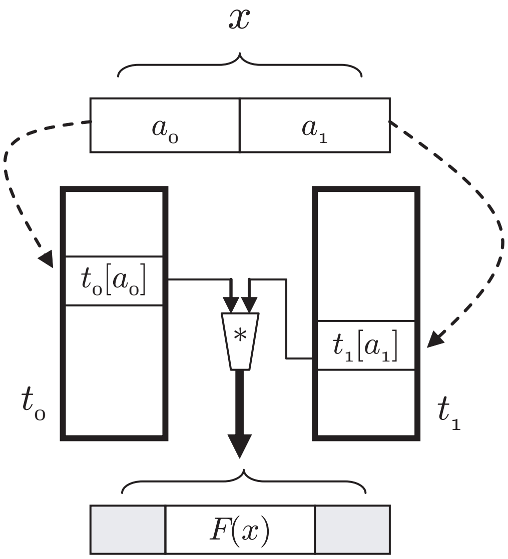


Fig. 3. A candidate function $F()$

$F()$, but the tables are a little harder to compute. These and other variations may be worth exploring further.

Assuming that we define $F()$ by letting $F(x) = G(t, x)$ for some function $G()$ (either by letting $F(a_0 \mid a_1) = \text{middle-bits } (t_0[a_0] * t_1[a_1])$ or in some other way), we may still use a trivial definition such as $F'(x) = F(x) \text{ xor } j$ to generate other functions, or we may generate other functions by varying t.

The definition $F(x) = G(t, x)$ can be generalized in useful ways. If $G()$ yields $n \times 2^b$ bits, where b is a small integer, we may apply $G()$ to a parameter t and to the $n - b$ high-order bits of x, then extract $F(x)$ from the result, as well as $F(x')$ for every x' that differs from x only in the b low-order bits. Interestingly, this definition makes the cost of applying $F()$ to all values in $0..(2^n - 1)$ be the cost of 2^n single applications divided by 2^b; this cost reduction helps in building a table for $F^{-1}()$.

5.2 Setting parameters

In order to instantiate our method, we need to pick values for various parameters (n, k, f, p, ...). These choices are constrained by the available technology, and they are informed by several preferences and goals. Next we discuss some settings for these parameters and their consequences; many other similar settings are possible. All these settings are viable with current machines, and they all lead to seconds or minutes of memory-bound work for S, as intended.

Suppose that we want the table for $F^{-1}()$ to fit in 32MB memories, but not in 8MB caches. These constraints determine the possible values of n to be 22 or 23. One might imagine that each entry in the table will take only 3 bytes, but such a compact encoding would probably be impractical. It is more realistic to allocate 4 or 6 bytes per entry to allow for collisions. With $n = 22$, a table for $F^{-1}()$ will occupy around 16MB (with 4 bytes per entry) or 24MB (more comfortably, with 6 bytes per entry). With $n = 23$, a table for $F^{-1}()$ will occupy around 32MB (with 4 bytes per entry) or 48MB (more comfortably, with 6 bytes per entry), so $n = 23$ may not be viable. In what follows, we proceed with $n = 22$ because that appears to be the appropriate value for current machines. We recommend increasing n as soon as cache sizes require it.

We have some choice in the cost f of applying $F()$, within the constraints of section 3.2. A larger value will result in more work for R if it sets problems or checks solutions by applying $F()$. A larger value should also result in more work for S if it adopts a CPU-intensive algorithm, so a larger value leaves room for a more expensive memory-bound solution (through a larger k). However, these effects cease when f reaches the cost r of a memory read on a fast machine, because S could replace many applications of $F()$ with lookups at that point. Thus S will pay at most r for applying $F()$ on average, perhaps much less with caching and other optimizations. In what follows, we consider three possible values for f on a fast machine: $f = r$, $f = r/2$, and $f = r/8$.

In light of constraints 1 and 2 of section 3.2, we should set the number k of iterations around 2^{12}. We have some freedom in the setting of k. A larger k will lead to more work per problem, for both parties S and R, but with a better (larger) ratio between the work of S and the work of R. Conversely, a smaller k will result in less work per problem, with a smaller work ratio. Therefore, we tend to prefer larger values for k. When k is too large, CPU-intensive solutions become competitive with the table-based approach, and their cost is not uniform across machines. When k is too small, the cost of building a table for $F^{-1}()$ becomes dominant in the table-based approach, and this cost is not necessarily uniform across machines. In what follows, we proceed with $k = 2^{13}$ if $f = r$, with $k = 2^{12}$ if $f = r/2$, and with $k = 2^{11}$ if $f = r/8$.

Finally, we have some choice in the number p of problems over which a table for $F^{-1}()$ should be amortized. Generally, a larger p is better, primarily because it gives us more freedom in setting other parameters. The number p could be huge if we used a fixed function (or a fixed master function) forever. However, it is prudent to use a different function for each problem, and also to change master functions at least from time to time. An obvious possibility is to group problems and to adopt a new master function for each group (see section 3.3). We can usually describe the master function concisely, by a short name plus the seed to a random number generator or a cryptographic key, in approximately 20 bytes. We can usually describe each derived function in 0–2 bytes. We can present each problem in 6 bytes (including the required checksum), and each solution in 3 bytes. For $p = 128$, each group of problems occupies up to 1KB, giving rise to a visible but reasonable communication cost. The communication cost can be drastically reduced with the non-interactive variant of section 4, if we so wish. For the sake of definiteness, we proceed with $p = 32$. Each group of 32 problems occupies only 192 bytes without function descriptions, and a little more with them.

We expect that a machine can do roughly 2^{23} reads per second from memory (within a small factor). On the basis of this data, we can calculate the cost of setting and solving problems:

—With $f = r$ and $k = 2^{13}$, we intend that S perform 2^{24} reads per problem, so S should take 2 seconds per problem.

The setting of a problem will require 2^{13} applications of $F()$, which will take one millisecond on a fast machine.

—With $f = r/2$ and $k = 2^{12}$, we intend that S perform 2^{22} reads per problem, so S should take .5 seconds per problem.

The setting of a problem will require 2^{12} applications of $F()$, which will take .25 milliseconds on a fast machine.

—With $f = r/8$ and $k = 2^{11}$, we intend that S perform 2^{20} reads per problem, so S should take .125 seconds per problem.

The setting of a problem will require 2^{11} applications of $F()$, which will take 32 microseconds on a fast machine.

When we multiply these costs by the number of problems (32), we obtain costs for solving groups of problems: 64, 16, and 4 seconds, respectively. We now check that these costs dominate the cost of building a table for $F^{-1}()$. The cost of building a table is roughly that of 2^{22} applications of $F()$ and writes. On a fast machine, the writes account for a substantial part of the cost; the cost should be under one second, in any case. On a slow machine, the applications of $F()$ account for most of the cost; the cost may go up considerably, but no higher than the cost of solving a group of problems. Even if each application of $F()$ were to cost as much as $16 \times r$ on a slow machine, building a table would take under 10 seconds. Thus, the total cost for building a table and solving a group of problems remains within a small factor across machines.

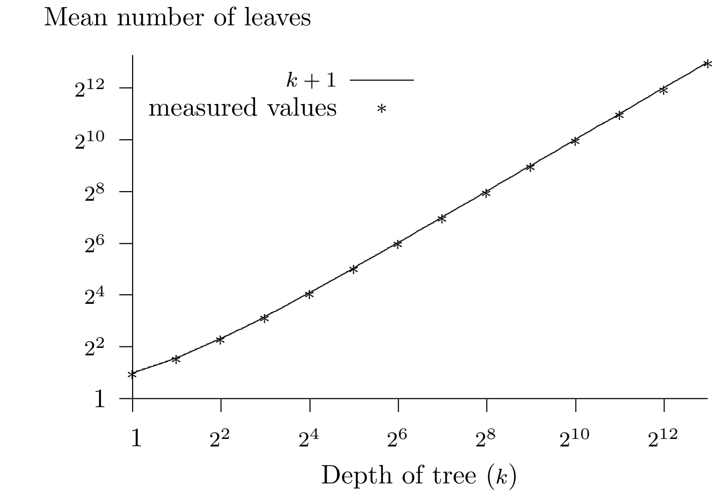
These costs compare favourably to those of solving problems with a CPU-intensive algorithm. Suppose that some CPU-intensive algorithm could solve each problem with just 4×2^n applications of $F()$, that is, with just 2^{24} applications of $F()$ (letting $c = 4$, in the notation of section 3.2). Depending on whether $f = r$, $f = r/2$, or $f = r/8$, those applications will cost as much as 2^{24}, 2^{23}, or 2^{21} reads, respectively. In comparison, the memory-bound approach requires 2^{24}, 2^{22}, and 2^{20} reads, respectively.

Relying on an 8MB cache and a compact encoding, S might be able to evaluate $F^{-1}()$ with only 4 applications of $F()$ (see section 3.2). Thus, S might replace each read with 4 applications of $F()$ and otherwise perform the same search as in the memory-bound approach. When $f = r$ or $f = r/2$, this strategy does not beat a CPU-intensive algorithm that could solve each problem with 2^{24} applications of $F()$, and a fortiori it does not beat the memory-bound algorithm. When $f = r/8$, this strategy may produce a solution at the same cost as 2^{19} reads, so it might appear to be faster than the memory-bound algorithm. However, the memory-bound algorithm will have that same cost if S has an 8MB cache and holds there half of a table for $F^{-1}()$ with a compact encoding.

6. EXPERIMENTS

In this section we report on several experiments related to our method. First, we give evidence for the claims about tree sizes made in section 3.2. Then we show that memory latencies vary far less than CPU speeds. Finally we show that the speed of our memory-bound functions varies significantly less across machines than the speed of CPU-bound functions proposed for similar purposes.

Tree sizes. For two functions on 22-bit integers, we found the mean number of leaves and nodes in trees formed using the procedure given in section 3.2. One of the functions was derived by calling the system (UNIX) random number generator 2^{22} times. The other was the function $F(a_0 \mid a_1) =$ middle-bits $(t_0[a_0] * t_1[a_1])$ discussed in section 5.1, where the elements of t_0 and t_1 were obtained from the system random number generator.



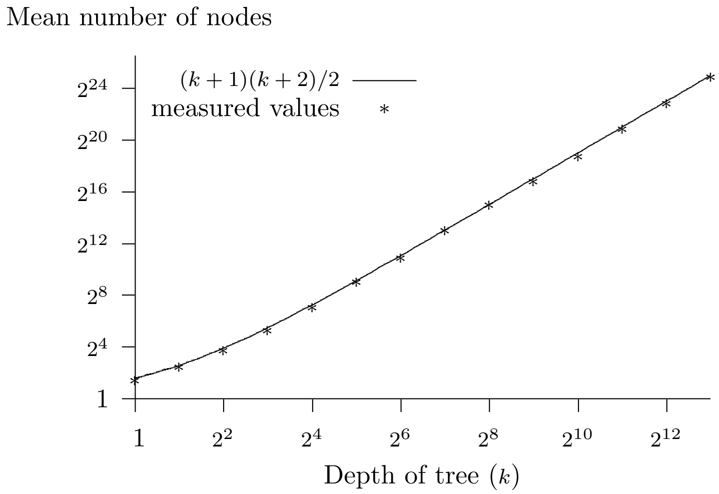


Fig. 4. Mean numbers of leaves and nodes in trees of depth k.

The results are indistinguishable for the two functions. We expect similar results for other pseudo-random functions.

We averaged over all possible starting points x_0, and varied the depth k of the trees. Figure 4 shows that the mean number of leaves in such trees closely matches $k + 1$, and that the mean number of nodes in such trees closely matches $(k + 1)(k + 2)/2$.

Timings. Next we give experimental results for five modern machines that were bought within a two-year period in 2000–2002, and which cover a range of performance characteristics. All of these machines are sometimes used to send e-mail—even the settop box, which is employed as a quiet machine in a home. Table I lists the machines; Table II gives their CPU clock frequencies, memory read times, and approximate prices.

We obtained the memory read times by measuring the time taken to follow a long linked

machine	model	processor type
server	Dell PowerEdge 2650	Intel Pentium 4
desktop	Compaq DeskPro EN	Intel Pentium 3
laptop	Sony PCG-C1VN	Transmeta Crusoe
settop	GCT-AllWell STB3036N	Nat. Semi. Geode GX1
PDA	Sharp SL-5500	Intel SA-1110

Table I. The machines used in our experiments.

machine	CPU clock frequency	memory read time	approximate price (US$)
server	2.4GHz	0.19μs	$3000
desktop	1GHz	0.14μs	$600
laptop	600MHz	0.25μs	$1000
settop	233MHz	0.23μs	$300
PDA	206MHz	0.59μs	$500

Table II. Machine characteristics.

machine	CPU-bound (HashCash)		memory-bound (trees)	
	seconds	ratio server=1	seconds	ratio desktop=1
server	110	1.0	24	1.1
desktop	140	1.3	22	1.0
laptop	330	3.0	42	1.9
settop	1430	13.0	91	4.1
PDA	1920	17.5	100	4.5

Table III. The performance of HashCash and of our tree searches on the machines listed in Table I. The absolute times are of less interest than the range of times for a given function.

list; the list entries were scattered widely through memory, and positioned so as to ensure that each access missed in the cache. Thus, these times include TLB miss overhead. This overhead is substantial on our PDA and it explains the high latency on that machine, where there seems to be an additional memory reference for most reads from the list. None of the machines have huge caches—the largest was on the server machine, which has a 512KB cache. Although the clock speeds of the machines vary by a factor of 12, the memory read times vary by a factor of only 4.2. This measurement confirms our premise that memory read latencies vary much less than CPU speeds.

Although the settop box might appear to have an attractive performance for its price, it is actually slower than its clock speed and memory access time might suggest, partly because it has a fairly simple pipeline. At the high end, the server has lower performance than one might expect, because of a complex pipeline that penalizes branching code. In general, higher clock speeds correlate with higher performance, but the correlation is far from perfect.

Table III shows the performance of a CPU-bound task (HashCash [Back 1997]) and of our memory-bound computations on the machines listed in Table I. The times are

machine	table build time seconds
server	0.9
desktop	1.1
laptop	3.2
settop	6.1
PDA	5.6

Table IV. Times to build the inverse table used in the memory-bound functions.

rounded to two significant figures. The HashCash times are for minting 100 20-bit HashCash tokens—that is, finding 100 independent 20-bit partial collisions in SHA-1. The memory-bound times are the means over 10 runs, each consisting of 128 depth-first tree searches, using the parameters $n = 22$ and $k = 2^{11}$. These results do not include the time taken to build the table for $F^{-1}()$, which we consider next.

Table IV shows the time taken to build the table for $F^{-1}()$. We used a straightforward implementation in which each insertion of an entry into the table requires at least one read for resolving collisions, followed by a write to store the entry. We let $F(a_0 \mid a_1) =$ middle-bits $(t_0[a_0] * t_1[a_1])$. Evaluating $F()$ is cheap compared to a memory access; thus, most of the cost of building the table is due to the memory accesses needed for insertions into the table. For this function, the cost f is under $r/8$, but increasing it to $r/8$ does not substantially affect these results. On each machine, the time taken to build the table is insignificant when compared with the corresponding number in Table III. The latter number corresponds to 128 problems. If instead each table were amortized over just 32 problems, building the table would contribute no more than 25% of the total time of solving a group of problems. In any case, the ratio across machines remains under 5.

The same executables were used on the desktop, laptop, and settop machines. The code was compiled with the Intel C compiler. The executables for the server machine were compiled with optimization for the Pentium 4; performance without this specialized optimization was poor. The executables for the PDA were compiled with gcc. We used less memory (20MB) for the inverse table on the PDA, in order to make it fit in the limited space available—the Sharp SL-5500 provides only 32MB of its memory to applications and the operating system. On the other machines, we used 24MB.

These experiments demonstrate several points. First, the effective performance of the machines varies more than clock speed alone might indicate. This variation is the result of the faster, more expensive processors having more elaborate pipelines. Second, the desktop machine is the most cost-effective one for both CPU-bound and memory-bound computations; in both cases, attackers are best served by buying the same type of machines as ordinary users. Finally, the memory-bound functions succeed in maintaining a performance ratio between the slowest and fastest machines that is not much greater than the ratio of memory read times.

The experiments also provide validation of the approximate calculations of section 5.2 in which we discuss settings for our parameters. In that section, we assume a machine with a memory read time of 2^{-23} seconds, while these real machines have somewhat slower memories. Once this difference is taken into account, the experimental results are largely consistent with the calculations.

7. CONCLUSIONS AND OPEN ISSUES

This paper is concerned with finding moderately hard functions that most recent computer systems will evaluate at about the same speed. Such functions can help in protecting against a variety of abuses. The uniformity of their cost across systems means that they need not inconvenience low-end, legitimate users in order to deter high-end attackers. We define and study a family of moderately hard functions whose computation can benefit crucially from accesses to a large table in memory. Our experimental results indicate that these memory-bound functions are much more egalitarian across machine types than CPU-bound functions.

It is possible that technology changes will result in more diverse memory systems in the future, and then memory-bound functions would no longer provide an egalitarian protection against abuses. However, we have identified several parameters that can be tuned as technology evolves. We have also found a number of ideas and tricks that should help in adapting our approach to different circumstances and applications.

The literature contains many papers that treat the space requirements of particular algorithms, cache-miss rates, and tradeoffs between time and space. Some of that work has been a source of inspiration for us in seeking memory-bound functions. In particular, we remembered the classic meet-in-the-middle attacks on double DES; using large tables, these attacks are much faster than naive CPU-intensive algorithms [Menezes et al. 1996]. However, these attacks can be implemented with multiple passes over the key space and smaller tables, so they are not necessarily limited by memory latency. We have not come across any previous results that we could directly exploit for our purposes, though we may still find some. More generally, it is desirable to investigate alternative memory-bound computations; some are studied in the recent work of Dwork, Goldberg, and Naor [Dwork et al. 2003].

The literature also contains some models of memory hierarchies (e.g., [Aggarwal et al. 1987]). An interesting subject for further work is to use such models in order to develop a foundation for memory-bound computations, if possible proving that particular computations (such as ours) are inherently memory-bound. Dwork, Goldberg, and Naor are making substantial progress in this direction, so far relying on “random oracles” [Dwork et al. 2003].

Many considerations may affect the acceptance of moderately hard functions, and of memory-bound functions in particular. The problems of large-scale deployment, such as software distribution and handling legacy systems, may be the most challenging. In addition, as the price of computer time falls, one must prescribe longer computations in order to impose a given cost. For example, in order to impose a cost of one cent (well under the current cost of physical bulk mail in the US), a computation of at least several minutes is required today; half an hour may be needed in the not-too-distant future. Moreover, memory-bound functions can interfere with concurrently running applications in a multi-tasking environment, both because they consume memory and because they can displace the applications’ code and data from caches. For these reasons, users may not tolerate moderately hard functions, not even egalitarian ones. On the other hand, even costs below one cent might be effective against some abuses, such as spam. Cache interference can be reduced by arranging that the inverse table map to a subset of the cache lines, and it can be avoided by accessing memory with instructions that bypass the caches. Furthermore, users may tolerate, and perhaps not even notice, long computations done asynchronously when their machines are otherwise idle. We rely on such asynchronous computations in an ongoing project [Abadi et al. 2003].

ACKNOWLEDGMENTS

We are grateful to Dan Simon for suggesting the function

$$F(a_0 \mid a_1) = \text{middle-bits } \ (t[a_0] * t[a_1])$$

where t is a table of random 32-bit primes. We also wish to thank Dave Conroy and Chuck Thacker for information on memory systems; Moni Naor, for explaining prior work on inverting functions; and Andrew Birrell, Cynthia Dwork, Andrew Goldberg, Michael Isard, Anna Karlin, and Adam Smith, for many interesting discussions.

REFERENCES

ABADI, M., BIRRELL, A., BURROWS, M., DABEK, F., AND WOBBER, T. 2003. Bankable postage for network services. In *Asian 2003*. Lecture Notes in Computer Science, vol. 2896. Springer, 72–90.

ABADI, M., BURROWS, M., MANASSE, M., AND WOBBER, T. 2003. Moderately hard, memory-bound functions. In *Proceedings of NDSS 2003 (Networks and Distributed Systems Security)*. 25–39.

ABADI, M., LOMAS, T. M. A., AND NEEDHAM, R. 1997. Strengthening passwords. SRC Technical Note 1997 – 033, Digital Equipment Corporation, Systems Research Center. September/December.

AGGARWAL, A., ALPERN, B., CHANDRA, A., AND SNIR, M. 1987. A model for hierarchical memory. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. 305–314.

AHN, L., BLUM, M., HOPPER, N. J., AND LANGFORD, J. 2003. CAPTCHA: Using hard AI problems for security. In *Advances in Cryptology – EUROCRYPT 2003*. Lecture Notes in Computer Science, vol. 2656. Springer, 294–311.

ASTLEY, M. R. 2001. Repost of private mail re: No free spam. Posting available on the web at URL `www.camram.org/mhonarc/spam/msg00030.html`.

BACK, A. 1997. HashCash. Available on the web at URL `www.cypherspace.org/˜adam/hashcash`.

CAMRAM 2002. Welcome to CAMRAM. Available on the web at URL `www.camram.org`.

CRANOR, L. F. AND LAMACCHIA, B. A. 1998. Spam! *Communications of the ACM 41,* 8 (Aug.), 74–83.

DWORK, C., GOLDBERG, A., AND NAOR, M. 2003. On memory-bound functions for fighting spam. In *Advances in Cryptology—CRYPTO 2003*. Lecture Notes in Computer Science, vol. 2729. Springer, 426–444.

DWORK, C. AND NAOR, M. 1992. Pricing via processing or combatting junk mail. In *Advances in Cryptology—CRYPTO '92*. Springer, 139–147.

FIAT, A. AND NAOR, M. 2000. Rigorous time/space trade-offs for inverting functions. *SIAM Journal on Computing 29,* 3 (June), 790–803.

FLAJOLET, P. AND ODLYZKO, A. 1990. Random mapping statistics. In *Advances in Cryptology – EUROCRYPT ' 89*, J.-J. Quisquater and J. Vandewalle, Eds. Lecture Notes in Computer Science, vol. 434. Springer, 329–354.

GABBER, E., JAKOBSSON, M., MATIAS, Y., AND MAYER, A. J. 1998. Curbing junk e-mail via secure classification. In *Financial Cryptography*. 198–213.

HELLMAN, M. E. 1980. A cryptanalytic time-memory trade off. *IEEE Transactions on Information Theory IT-26,* 4, 401–406.

JAKOBSSON, M. AND JUELS, A. 1999. Proofs of work and bread pudding protocols. In *Proceedings of the IFIP TC6 and TC11 Joint Working Conference on Communications and Multimedia Security (CMS '99)*. Kluwer.

JUELS, A. AND BRAINARD, J. 1999. Client puzzles: A cryptographic defense against connection depletion. In *Proceedings of NDSS '99 (Networks and Distributed Systems Security)*. 151–165.

MANBER, U. 1996. A simple scheme to make passwords based on one-way functions much harder to crack. *Computers & Security 15,* 2, 171–176.

MAY, T. C. 1993. Timed-release crypto. Unpublished manuscript.

MENEZES, A. J., VAN OORSCHOT, P. C., AND VANSTONE, S. A. 1996. *Handbook of Applied Cryptography*. CRC Press.

MERKLE, R. C. 1978. Secure communications over insecure channels. *Communications of the Association for Computing Machinery 21,* 4 (Apr.), 294–299.

NAOR, M. 2002. Private communication.

OECHSLIN, P. 2003. Making a faster cryptanalytic time-memory trade-off. In *Advances in Cryptology—CRYPTO 2003*. Lecture Notes in Computer Science, vol. 2729. Springer, 617–630.

QUISQUATER, J.-J., STANDAERT, F.-X., ROUVROY, G., DAVID, J.-P., AND LEGAT, J.-D. 2003. A cryptanalytic time-memory tradeoff: First FPGA implementation. In *Cryptographic Hardware and Embedded Systems – CHES 2002*. Lecture Notes in Computer Science, vol. 2523. Springer, 593–609.

RIVEST, R. L., SHAMIR, A., AND WAGNER, D. A. 1996. Time-lock puzzles and timed-release crypto. Technical Report MIT/LCS/TR-684, MIT. Feb.

Received July 2003; December 2003; accepted ???