Once Upon a Time-Memory Tradeoff

Mark Stamp

July 26, 2003

1 Introduction

Once upon a time, I worked for a small startup company. The pay was excellent, but the hours were long and vacation time was limited. I currently work as a college professor, where the pay is not-so-good but the hours and vacation time are excellent. For several years, I also worked for the government, where the pay is reasonably good and the hours and vacation time are reasonably good. So, it could be argued that the optimal "time-money tradeoff" is attained with a government job[1].

A different sort of balancing act occurs with a time-memory tradeoff (TMTO). A TMTO attempts to balance one-time work (the result of which is stored in "memory") with the time required to run the algorithm. A TMTO is not an algorithm *per se*, but instead it's a general technique that can be applied to improve the performance of many different algorithms. Usually, a TMTO is developed to improve the speed of an algorithm by utilizing one-time work, which results in increased storage (memory) requirements when the resulting algorithm is executed. Of course, it is also possible to work in the opposite direction by reducing the one-time work at the expense of more work each time the algorithm is executed. The goal is to balance the one-time work (memory) requirement with the speed of the algorithm (time). In some cases, the time versus memory tradeoff is obvious, but in some cases it is not so obvious.

In this paper we consider three different applications of a TMTO. The first example, $\texttt{popcnt}(x)$, is very simple, but illustrates the basic concept. The second example, Shank's algorithm, is slightly less obvious, while the final example, Hellman's cryptanalytic TMTO, is even less obvious—and also happens to be one of the most celebrated examples of a TMTO.

2 A very simple example

Given a non-negative integer x, its "population count", or $\texttt{popcnt}(x)$, is defined as the number of ones in the binary expansion of x. For example,

$$\texttt{popcnt}(13) = 3$$

[1]Personally, I like teaching, which changes the equation considerably—much to my wife's chagrin. Apparently, there's no accounting for human nature.

since 13 is, in binary, 1101.

Perhaps the most obvious way to compute popcnt(x) is given below, where len(x) is the number of bits used to store the integer x, "`>>`" is the right shift and "`&`" is a binary AND.

```
// compute t = popcnt(x)
t = 0
for i = 0 to len(x) − 1
    t = t + (x >> i) & 1
next i
```

This approach requires len(x) operations and essentially no memory.

We might be able to obtain a faster algorithm for popcnt(x) if we are willing to employ a pre-computation and use some memory to store the pre-computation results. For example, we could pre-compute popcnt(y) for each $y \in \{0, 1, \ldots, 255\}$, and store the resulting values in an array, say, $p[y]$. Then popcnt(x), where, for example, x is a 32-bit integer, could be computed as

```
// compute t = popcnt(x) for a 32-bit integer
t = p[x & 0xff] + p[(x >> 8) 0xff] + p[(x >> 16) 0xff] + p[(x >> 24) 0xff]
```

where "`0xff`" hexadecimal `ff`. This approach requires 4 operations per popcnt(x), along with the one-time work of computing $p[y]$ and storage (or memory) of size 256.

We could instead choose to pre-compute popcnt(y) for all $y \in \{0, 1, \ldots, 15\}$, in which case we would need 8 operations for each popcnt(x) (assuming a 32-bit x) while using only 16 memory elements. The logical choices for time versus memory are summarized in Table 1. Of course, the optimal "tradeoff" between the lookup table size and number of operations will depends on many factors, including the specific architecture, but could easily be determined by experimentation.

memory	operations
0	32
2^2	16
2^4	8
2^8	4
2^{16}	2
2^{32}	1

Table 1: TMTO for popcnt(x) for 32-bit integers

3 A not-quite-so-simple example

Let p be a prime. Suppose we can find $g \in \{1, 2, \ldots, p-1\}$ such that for any $n \in \{1, 2, \ldots, p-1\}$, there exists some k such that $n = g^k \bmod p$. Then g is a *generator* of $\{1, 2, \ldots, p-1\}$, since any element in the set can be obtained as a power of g.

It follows that for any $m \in \{1, 2, \ldots, p-1\}$ there exists a unique $e \in \{1, 2, \ldots, p-2\}$ such that $m = g^e \bmod p$. The exponent e is know as the *discrete logarithm of m to the base g*. When the base is clear from context, we simply refer to the *discrete log* of m. We use the familiar notation

$$e = \log_g(m).$$

The discrete logarithm arises in many situations. For example, the well-known Diffie-Hellman key exchange algorithm can be broken by solving a discrete logarithm problem [8].

Given p, g and $m \in \{1, 2, \ldots, p-1\}$, we want to find $e \in \{1, 2, \ldots, p-2\}$ such that

$$m = g^e \bmod p. \tag{1}$$

One way to find e is to simply try each possible value $e = 1, 2, \ldots, p-2$ until a solution to (1) is found. This would require, on average, a work factor of $(p-2)/2$ and virtually no storage (memory). However, there is a better approach that relies on a TMTO.

For any $x \in \{1, 2, \ldots, p-2\}$, the value of e in (1) satisfies $e/x = i + j/x$, or $e = xi + j$, for some unique $i \leq e/x$ and $j < x$. Shank's algorithm [7] makes use of this simple observation to yield a TMTO for the discrete logarithm problem.

In Shank's algorithm, we first pre-compute a list L_r as follows.

1a. Let $r = \lceil \sqrt{p-1} \rceil$

2a. Compute $g^{rj} \bmod p$ for $j = 0, 1, \ldots, r-1$

3a. Let L_r be the list obtained by sorting the ordered pairs $(j, g^{rj} \bmod p)$ on the second coordinate

Then given any $m \in \{1, 2, ..., p-1\}$, we compute a list L_m in the following manner.

1b. Compute $mg^{-i} \bmod p$ for $i = 0, 1, \ldots, r-1$

2b. Let L_m be the list obtained by sorting the ordered pairs $(i, mg^{-i} \bmod p)$ on the second coordinate

Finally, the discrete log, $\log_g(m)$, can be found as follows.

1c. Find an element of L_r and an element of L_m that agree in their second coordinates, say, $(j, x) \in L_r$ and $(i, x) \in L_m$

2c. Then $\log_g(m) = rj + i \bmod (p-1)$

The correctness of the result in 2c. follows since $g^{rj} = mg^{-i} \bmod p$ and hence by basic properties of exponentiation $g^{rj+i} = m \bmod p$. The $\bmod (p-1)$ in 2c. follows from Fermat's Little Theorem [2].

Note that if we had chosen a different value for r, the algorithm would still work, but the lists L_r and L_m would be different sizes. By choosing $r = \lceil \sqrt{p-1} \rceil$, the memory requirement is about $2\sqrt{p-1}$ and the time requirement is about $2\sqrt{p-1}$, neglecting the time required to sort the lists and the comparisons required to finding the matching list elements.

For example, suppose $p = 257$ and $g = 3$. Then $r = 16$ and the sorted list L_r is found to be

(0,1)	(3,2)	(6,4)	(9,8)
(12,16)	(15,32)	(2,64)	(5,128)
(13,129)	(10,193)	(7,225)	(4,241)
(1,249)	(14,253)	(11,255)	(8,256).

For $m = 132$, the sorted list L_m is

(9,23)	(1,44)	(3,62)	(5,64)
(8,69)	(12,77)	(15,79)	(6,107)
(0,132)	(10,179)	(2,186)	(4,192)
(13,197)	(7,207)	(11,231)	(14,237).

The pairs $(2, 64)$ from L_r and the pair $(5, 64)$ from L_m match in the second elements. Therefore, the solution is given by

$$\log_3(132) = 2 \cdot 16 + 5 = 37$$

and it is easily verified that $3^{37} = 132 \bmod 257$.

Notice that if we want to find another discrete logarithm with the same g and p, we only need to recompute the list L_m, not the list L_r. For example, to find $\log_3(200)$, we compute L_m (sorted on the second component) as

(10,22)	(12,31)	(14,32)	(3,55)
(9,66)	(7,80)	(11,93)	(13,96)
(4,104)	(2,165)	(15,182)	(8,198)
(0,200)	(5,206)	(1,238)	(6,240).

from which we find the pairs $(15, 32)$ and $(14, 32)$ from L_m, giving $\log_3(200) = 15 \cdot 16 + 14 = 254$. Therefore, if we have many discrete logarithms to compute for a fixed g and p, we can amortize the cost of computing L_r, and hence the cost of each discrete logarithm is dominated by $\sqrt{p-1}$. This is a significant improvement over the work factor of $(p-2)/2$ required by the naïve approach.

The current best-known method for computing discrete logarithms is Pollard's rho algorithm. The article [5] describes the basic algorithm as well as extensions.

4 An even-less-simple example

Hellman—of Diffie-Hellman fame—describes a cryptanalytic TMTO attack in [4]. Before discussing the attack, we first give a brief description of the necessary cryptographic background.

A *block cipher* can be viewed as a function that takes a *plaintext* block of n bits and a *key* of k bits as input, producing an n-bit *ciphertext*. This process is known as *encryption*. The process must be invertible, so the block cipher must also be capable of combining the ciphertext together with the key (the same key used for encryption) to recover the plaintext. This inverse process is known as *decryption*.

For a given block cipher, let E be the encryption function, D the corresponding decryption function, K a key, P a plaintext block and C the corresponding ciphertext block. Then P and C are each n bits in length, while the key K is k bits. We write

$$C = E_K(P) \quad \text{and} \quad P = D_K(C).$$

Encryption and decryption are illustrated in Figure 1.

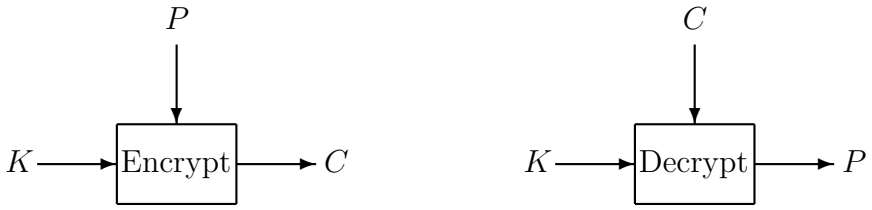

Figure 1: Block Cipher

An attacker's goal is to recover the key K. If the attacker only has access to ciphertext, then he must conduct a ciphertext-only attack. On the other hand, if he knows the plaintext that corresponds to ciphertext, he is in the more advantageous situation of a known-plaintext attack[2]. Even more advantageous—from the attacker's perspective—is a chosen-plaintext attack, where the attacker is able to choose some amount of plaintext and obtain the corresponding ciphertext. In the remainder of this section, we consider a chosen-plaintext attack.

Assuming there is no known weakness in the block cipher algorithm, perhaps the most obvious attack is to simply try decrypting the ciphertext C with each possible key K until the (chosen) plaintext P appears. Since there are 2^k possible keys, and we expect to find K after trying half of the keys, the expected work is on the order of 2^{k-1}, while the memory requirement is negligible. At the other extreme, we could precompute the ciphertext C for each possible key for the given (chosen) plaintext P. This would require one-time work of 2^k

[2]You might wonder why any attack is necessary if the attacker already knows the plaintext. The assumption is that the attacker knows *some* of the plaintext and by recovering the key he will obtain access to *all* of the plaintext that was encrypted with the recovered key.

and 2^k storage, but then each time we run the attack, only a single table lookup would be required. The one-time work could be amortized over the number of number of times that the attack is conducted. Of course, this assumes that the attacker is always able to choose identical plaintext.

Hellman's TMTO attack finds a middle ground between these two extremes. The attack requires some one-time work, producing a table of results. This table (the memory part of the TMTO) is then used in order to reduce the amount of work required (the time part of the TMTO) in any particular attack.

To illustrate Hellman's TMTO consider the following idealized example. Suppose we have a block cipher with block size $n = 64$ bits and key size $k = 64$. Since the key is 64 bits, there are 2^{64} distinct keys—said to be equal to the number of blades of grass on earth. Since this is a chosen plaintext attack, we choose plaintext P and obtain the corresponding ciphertext $C = E_K(P)$. The challenge is to recover the unknown key K.

Let $K_0 \in \{0,1\}^{64}$ be a possible key value. We denote K_0 as SP, the "starting point" of a chain of t encryptions and K_{t-1}, as EP, the "ending point". Such a chain can be computed as

$$\begin{aligned} SP &= K_0 \\ K_1 &= E_{SP}(P) \\ K_2 &= E_{K_1}(P) \\ K_3 &= E_{K_2}(P) \\ &\vdots \\ EP &= K_{t-1} = E_{K_{t-2}}(P). \end{aligned}$$

Figure 2 illustrates such a chain.

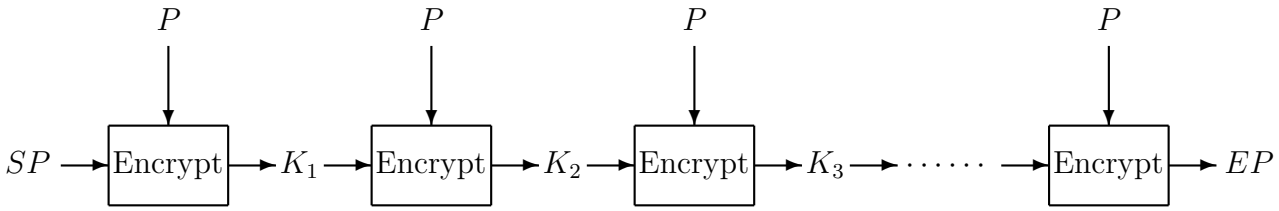

Figure 2: Chain of encryptions

Suppose that $t = 2^{32}$ and, further, suppose that we can find $m = 2^{32}$ starting points such that none of the resulting chains (each of length t) overlap[3]. Then each of the 2^{64} possible key values appears within one—and only one—chain.

[3]This is totally unrealistic, but it allows us to easily illustrate the TMTO concept. We return to reality below.

The "memory" part of the TMTO is implemented by storing only the starting points and ending points of the chains, namely,

$$(SP_0, EP_0), (SP_1, EP_1), (SP_2, EP_2), \ldots, (SP_{m-1}, EP_{m-1}).$$

In this imaginary example, $m = 2^{32}$ and hence the storage requirement—in terms of 64-bit words—is $2m = 2^{33}$.

The "time" part of the TMTO is implemented as follows. The attacker knows C and the chosen value P. He begins computing the chain

$$\begin{aligned}
\tilde{K}_0 &= C \\
\tilde{K}_1 &= E_{\tilde{K}_0}(P) \\
\tilde{K}_2 &= E_{\tilde{K}_1}(P) \\
\tilde{K}_3 &= E_{\tilde{K}_2}(P) \\
&\vdots \\
\tilde{K}_{t-1} &= E_{\tilde{K}_{t-2}}(P).
\end{aligned}$$

where at each step $i = 0, 1, \ldots, t-1$, the attacker compares $\tilde{K}_i$ to all of the endpoints, $EP_0, EP_1, \ldots, EP_{m-1}$. Since C is a possible key value, it must lie somewhere within one (and only one) chain. Therefore, for some $i \in \{0, 1, \ldots, t-1\}$ we must have $\tilde{K}_i = EP_j$, where $j \in \{0, 1, \ldots, m-1\}$. Then given this i and j, he will reconstruct the chain starting with SP_j,

$$\begin{aligned}
K_0 &= SP_j \\
K_1 &= E_{K_0}(P) \\
K_2 &= E_{K_1}(P) \\
K_3 &= E_{K_2}(P) \\
&\vdots \\
K_{t-1} &= EP_j = E_{K_{t-2}}(P).
\end{aligned}$$

Provided $\tilde{K}_0 \neq SP_j$ (which is easily avoided by not selecting C as a starting point) then for some $\ell \in \{1, 2, \ldots, t-1\}$ we must have $K_\ell = \tilde{K}_0 = C = E_K(P)$. Then since $K_\ell = E_{K_{\ell-1}}(P)$, the attacker has found the desired key $K = K_{\ell-1}$.

Note that the pre-computation phase requires on the order of $2^t 2^m = 2^{64}$ work. Having paid this enormous initial price, then each time the attack is run, on average only $2^{t-1} = 2^{31}$ encryptions will be required before an endpoint is found, followed by another $2^{t-1} = 2^{31}$ encryptions until C is found, giving a total work factor of about $2^t = 2^{32}$. If the attack is only to be executed once, a straightforward exhaust will find K with an expected work of 2^{63}, in which case it would make no sense to pre-compute the paths. However, it the attack is to be conducted many times, the pre-computation work can be amortized over the number of attacks, while the work of 2^{32} per attack is, in comparison, negligible.

Alternatively, we could compute paths that only cover a part of the key space, thereby reducing the pre-computation work. Then the probability of successfully finding an unknown key is equal to the proportion of the key space that is covered. This is the way that a cryptanalytic TMTO attack must be implemented since there is no feasible method to assure non-overlapping chains.

The fatal flaw in the TMTO discussed above is the assumption that we can find non-overlapping chains that partition the key space. When we generate a chain of encryptions, there are two bad things that can happen. An individual chain can overlap with itself, or two chains can merge into a single chain. Assuming that each new chain element is randomly selected from the set of possible keys, it is clear that the more of the space that is already covered, the more likely that a new chain will overlap with some previous chain.

Therefore, the goal of a cryptanalytic TMTO is to cover some percentage of the key space with chains. Then the resulting TMTO attack will recover any key that is in some chain. However, any key that does not appear in a chain will not be found with the TMTO attack. The attack is therefore probabilistic and the objective is to maximize the probability of success for a given amount of work.

In the simplest case, the key length k is equal to the cipher block length n and the algorithm can be described as follows.

// Find (SP_i, EP_i), $i = 0, 1, \ldots, m-1$
for $i = 0$ to $m-1$
    Generate a starting point SP_i
    $K_0 = SP_i$
    for $j = 1$ to t
        $K_j = E_{K_{j-1}}(P)$
    next j
    $EP_i = K_t$
next i

Notice that this process yields at most mt distinct predecessors. If the key K is among these predecessors, it will can be found by the attack discussed below.

Given $C = E_K(P)$, the attack proceeds as follows.

// Search for K within the pre-computed chains
$Y = E_C(P)$
for $i = 0$ to $t-1$
    for $j = 0$ to $m-1$
        if $Y == EP_j$ then
            Find K in the chain beginning with SP_j
            return(K)
        end if
    next j

$Y = E_Y(P)$
next i
return(key not found)

If $k \neq n$ then the situation is slightly more complex. In this case, we cannot compute chains as described above since we can't directly use the ciphertext as a key.

For concreteness, consider the Data Encryption Standard (DES), which has a key length of $k = 56$ and a block size of $n = 64$. We define a function $f(x)$ from the space of cipher blocks to key blocks by

$$f(x_0, x_1, \ldots, x_{63}) = (x_{i_0}, x_{i_1}, \ldots, x_{i_{55}})$$

where the indices $i_j \in \{0, 1, \ldots, 63\}$ are distinct. We use the function f to reduce a cipher block down to the size of a key which allows us to create chains as before. For example, given a starting point SP and a function f we can create a chain of length $t + 1$ by

$$\begin{aligned} K_0 &= SP \\ K_1 &= f(E_{K_0}(P)) \\ K_2 &= f(E_{K_1}(P)) \\ K_3 &= f(E_{K_2}(P)) \\ &\vdots \\ K_t &= EP = f(E_{K_{t-1}}(P)). \end{aligned}$$

The TMTO attack is similar to the previous case. For simplicity, suppose we are given P and $C = E_K(P)$ and only one pair (EP, SP). Then we compute

$$\begin{aligned} \tilde{K}_0 &= f(C) \\ \tilde{K}_1 &= f(E_{\tilde{K}_0}(P)) \\ \tilde{K}_2 &= f(E_{\tilde{K}_1}(P)) \\ \tilde{K}_3 &= f(E_{\tilde{K}_2}(P)) \\ &\vdots \\ \tilde{K}_t &= f(E_{\tilde{K}_{t-1}}(P)) \end{aligned}$$

looking for an index j such that $\tilde{K}_j = EP$. If such a j is found it follows that

$$f(C) = K_{t-j} = f(E_{K_{t-j-1}}(P). \tag{2}$$

However, unlike the $n = k$ case, it does not necessarily follow from (2) that $K = K_{t-r-1}$, since there are several possible predecessors. In other words, false alarms are possible, and the false alarm rate will depend on the size of $n - k$.

In practice, several different f functions will be used in order to increase the chances of covering the key space more uniformly and thus reducing the chance of collisions between—and within—chains. The pre-computation process can be implemented as given below.

```
// Find (SP, EP) pairs
for i = 0 to r − 1
    Generate a function f_i
    for j = 0 to m − 1
        Generate a starting point SP_j^i
        K_0 = SP_j^i
        for k = 1 to t
            K_k = f_i(E_{K_{k−1}}(P))
        next k
        EP_j^i = K_t
    next j
next i
```

Notice that in this process we find rm chains, each of length $t+1$. This yields rmt predecessors, any of which could lead to a given key K. If all of these rmt predecessors are distinct, then the chance of finding a randomly-selected key is about $rmt/2^k$.

For efficient searching, the pairs (SP_j^i, EP_j^i) for $j = 0, 1, 2, \ldots, m-1$, should be sorted on EP_j^i. Then given $C = E_K(P)$ and the pre-computed (SP, EP) pairs, the attack proceeds as follows.

```
// Search for K within pre-computed chains
for i = 0 to r − 1
    Y = f_i(E_C(P))
    for j = 0 to t − 1
        for k = 0 to m − 1
            if Y == EP_k^i then
                Find K_{t−i−1} in the chain from SP_k^i
                if K == K_{t−i−1} then // test via trial decryption
                    return(K)
                else
                    false alarm
                end if
            end if
        next k
        Y = f_i(E_Y(P))
    next j
next i
```

Note that this algorithm also works for the case where $n = k$, except that no false alarms will occur.

It is also possible to use "distinguished points" to obtain a slightly different version of the algorithm [1]. This variant might be preferable in certain distributed attack scenarios.

In general, we require $\ell = \lceil k/n \rceil$ matched plaintext/ciphertext pairs in order to uniquely determine a key of length k from cipher blocks of length n. So far, we have only considered $\ell = 1$, in which case a single chosen plaintext (and its corresponding ciphertext) suffices. In the case where $\ell > 1$, we need multiple chosen plaintext/ciphertext pairs. We can easily handle this case by simply defining a function f on multiple ciphertext blocks as

$$f(E_Y(P_0), E_Y(P_1), \ldots, E_Y(P_{\ell-1})) = f(x_0, x_1, \ldots, x_{n\ell} - 1) = (x_{i_0}, x_{i_1}, \ldots, x_{i_{k-1}})$$

and making the obvious modifications to the algorithms above.

Finally, it is important to balance the work factor with the probability of finding a key. The mathematics is beyond the scope of this article, but under reasonable assumptions, it is shown in [6] that it is best to select

$$m = t = r = 2^{k/3},$$

which yields a pre-computation work factor of 2^k and a probability of success of about 0.55.

References

[1] J. Borst, B. Preneel and J. Vandewalle, On the time-memory tradeoff between exhaustive key search and table precomputation, 19th Symposium on Information Theory in the Benelux, 1998,
`http://www.esat.kuleuven.ac.be/~borst/downloadable/tm.ps.gz`

[2] Fermat's Little Theorem,
`http://www.utm.edu/research/primes/notes/proofs/FermatsLittleTheorem.html`

[3] A time-memory tradeoff using distinguished points: new analysis & FPGA results, S. Francois-Xavier, R. Gael, Q. Jean-Jacques, L. Jean-Didie
`http://www.dice.ucl.ac.be/crypto/publications/2002/FPL2002_TMT.pdf`

[4] M. Hellman, A cryptanalytic time-memory tradeoff, *IEEE transactions on Information Theory*, Vol. 26, 1980, pp. 401–406.

[5] F. Kuhn and R. Struik, Random walks revisited: extensions of Pollard's rho algorithm for computing multiple discrete logarithms, Selected Areas in Cryptography: 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16-17, 2001, Lecture Notes in Computer Science (LNCS) 2259, Springer–Verlag Heidelberg, 2001, pp. 212–229, `http://distcomp.ethz.ch/publications/sac01.pdf`

[6] K. Kusuda and T. Matsumoto, Optimization of time-memory trade-off cryptanalysis and its application to DES, FEAL-32 and Skipjack, *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science*, E79–A, 1996, pp. 35–48, `http://search.ieice.or.jp/1996/pdf/e79-a_1_35.pdf`

[7] D. R. Stinson, *Cryptography: Theory and Practice*, CRC Press, 1995.

[8] What is Diffie-Hellman? RSA Laboratories Cryptography FAQ, July 2003, `http://www.rsasecurity.com/rsalabs/faq/3-6-1.html`