Secure Applications of Low-Entropy Keys

John Kelsey Bruce Schneier Chris Hall
Counterpane Systems
101 E. Minnehaha Parkway
Minneapolis, MN 55419
{kelsey,schneier,hall}@counterpane.com

David Wagner
U.C. Berkeley
Soda Hall, C.S. Division
Berkeley, CA 94720-1776
daw@cs.berkeley.edu

Abstract. We introduce the notion of key stretching, a mechanism to convert short s-bit keys into longer keys, such that the complexity required to brute-force search a $s+t$-bit keyspace is the same as the time required to brute-force search a s-bit key stretched by t bits.

1 Introduction—Why Stretch a Key?

In many real-world cryptographic systems, we are, for various reasons, limited to encryption keys with relatively low-entropy. In other words, we are limited to a key that is unlikely to take on more than (say) 2^{40} different values. This can happen for a variety of reasons: legacy systems that force the use of outdated 56- or 64-bit key lengths, keys that must be remembered and typed in by users, keys generated from realtively poor random bit sources, or various countries' restrictions on key-length.

In the realm of user-entered keys and passphrases, it has become common to carry out some kind of computationally expensive calculation to derive the actual key or secret value. This is intended to increase the cost of exhaustive search against a large portion of the possible keyspace. One problem with these ad hoc methods is that it's not always clear how much additional security they add. We are interested in explicitly answering a question of "how hard is this brute-force search," in terms of either dollars or bits.

In this paper, we describe a general kind of procedure, called *key stretching*, which adds a known number of bits to the expected difficulty of an exhaustive search attack. That is, we can make a 40-bit search as expensive as a 56-bit search would normally be.

The remainder of the paper is organized as follows. In Section 2 we describe a general framework for key-stretching algorithms. In Section 3 we describe various classes of attack that are possible on key-stretching algorithms, and some design principles that allow us to resist these attacks. In Section 4 we describe two different key stretching algorithms: one based on a one-way hash function, and the other on a block cipher. We are able to give proofs, under some reasonable assumptions, that any attacks on the hash-based scheme translate into attacks on the underlying cryptographic primitive. Finally, in Section 5, we discuss extensions to frustrate brute-force attacks which make use of massively parallel devices.

2 A General Model for Key Stretching

Key stretching is about adding a specific number of bits to a keysearch or brute-force attack of some kind. In a cryptographic context, what we'd like is for the following two things to be equally difficult:

A brute-force search of a key-stretching scheme that stretches a s-bit key to $s + t$ bits, and
A keysearch of the actual cipher, with $s + t$ bits of key.

All of our methods for doing this involve finding some function $F()$ which approximates a random function and which requires roughly 2^t work to compute. We must then convince ourselves that there are no shortcuts for the attacker, and that this function is roughly as hard for him to compute as it is for a user.

A general method for key-stretching is as follows:

1. We start with some low-entropy key, K_{short}. We want to derive a hard-to-search key, K_{long}, so that searching all possible values of K_{long} takes 2^{s+t} rekeyings and trial encryptions, where s is the number of bits of entropy of K_{short}, and t is the number of bits of stretching being done.
2. Let $Salt$ be an occasionally changing salt value. For some applications, $Salt$ changes once every year or few months. For others, it may never change. For still others, it changes with each new message.
3. Let $F()$ be some function that approximates a random function. We require that computing $F()$ is guaranteed to take about the same amount of work as searching 2^t keys in a brute-force exhaustive keysearch.
4. Let $K_{long} = F(K_{short}, Salt)$.

Once we have this algorithm, we must then convince ourselves that an attacker carrying out a keysearch of the stretched key has to carry out nearly all 2^t computations for each short key stretched to a long key. That is, there must be no significant "economies of scale" in doing the keysearch.

2.1 Salt

In Section 3, various precomputation attacks on key-stretching schemes are discussed. To resist these, we use a salt. The purpose of a salt value is to make it impossible for an attacker to precompute a list of all or a large portion of the possible long keys used for all messages.

There are three ways we can use salt values:

1. *Per System.* If we have many noninteroperable systems, we can use a different salt for each, perhaps based on the hash of the programmer's full name, the program's or system's name, and the date on which the code was completed. This forces the attacker to do a separate precomputation for each system attacked. If 2^{s+t} work-factor searches are sufficiently expensive, this may, in practice, mean that only a few such systems are ever attacked.
2. *Per Time Period.* For interoperable systems, we may wish to use a salt that changes periodically and unpredictably. For example, we might have a salt that changes once per year, and that is defined as the hash of all the stock prices of the stocks in the S&P 100, on the last active trading day in November. This is unpredictable by any attacker, so precomputations can't start until November for the next year's messages. Again, this works well if $s + t$ bit keysearches are sufficiently expensive.
3. *Per Message.* We can make precomputation attacks much less rewarding for passive eavesdroppers by changing the salt for each message sent, and deriving the salt randomly. Unfortunately, this makes a chosen-ciphertext attack possible, in which a precomputation with any salt allows the recovery of a given message's long key. This works because the attacker can simply replace the sender's salt with his own salt.

A *Hybrid* scheme is the best way to do this. If the system can handle the additional requirements, use two different values to derive each message's salt: a per-message random salt component, along with a long-term salt which is either changed each year or is set to different values for different non-interoperable systems. This forces the attacker to do a new precomputation each year (or for each noninteroperable system), in order to be able to successfully use even an active attack.

In our general scheme, described in this section, and in our two schemes described in Section 4, the salt value used is assumed to be the hash of whatever inputs were chosen for the salt value.

3 Attacks on Key Stretching

3.1 Economies of Scale

Keysearch attacks typically benefit from certain "economies of scale": optimizations that are not available or useful to an ordinary user, but that make an attacker's job easier. For example, in a straightforward brute-force search of DES [NBS77] with known plaintext and ciphertexts, an attacker can omit computation of the last round of DES in each trial encryption, optimize the order of the keys searched to minimize the work involved in changing keys, and build massively parallel hardware useful only for DES key searching.

In designing a key-stretching algorithm, the most important economies of scale attacks are the following:

Narrow-Pipe Attacks A "narrow-pipe" attack occurs when some intermediate stage of the key-stretching computation has too few bits of state. It is obvious that having an intermediate state of less than $s+t$ bits will leave us with less security than we expect, because an attacker can just guess this intermediate state. As will be described below, however, repeated intermediate values can also give an attacker economies of scale: he can reuse the repeated results. If any intermediate stages have fewer than $s+t+6$ bits of state, then this becomes an issue.[1]

Reuse of Computed Values In the generic construction given above, to make any strong statements about how much difficulty is added to a keysearch attack, we must know whether or not an attacker ever gets to reuse final or intermediate $F()$ computations. An attacker who can reuse these computations may be able to do a keysearch much more cheaply than we would otherwise expect. The issue is making sure that the intermediate computations have enough bits of state, as discussed above. However, we also need to review our specific choice of $F()$ to ensure that it can't be sped up substantially by reuse of intermediate values.

Special-Purpose Massively Parallel Keysearch Machines In some keysearch attacks, massively parallel computers are built to cheaply try all possible keys. Wherever possible, our choice of $F()$ should be influenced by the desire to make such attacks more expensive without making the function significantly more expensive for legitimate users on general-purpose computers to compute.

3.2 Cherry Picking

Even if the whole keysearch is guaranteed to take up at least (say) 2^{80} work, it may be that some part of the keyspace can be searched very cheaply. If this is the case for a significant portion of the keyspace in our key stretching scheme, then this becomes an important weakness. Just as it should cost 2^{72} trial encryptions and rekeyings to search 2^{-8} of all possible 80-bit keys, a 56-bit key stretched to 80 bits of effective strength should cost 2^{72} encryptions and rekeyings to search any 2^{-8} of the keyspace. In practice, this means that nearly all short keys should take about the same number of actual operations to stretch.

3.3 Nonrandom Long Keys

Ideally, the long keys should be indistinguishable from random bit strings for anyone who doesn't know the corresponding short keys. If this isn't the case, then some attacks may be possible based on guessing the long keys directly from these patterns. In particular, if there is some relationship between pairs (or k-tuples) of short keys that leads to some relationship between the corresponding pairs

[1] There's nothing magical about the number 6, it just ensures less than one percent reuse of the intermediate values. Note that this concern applies only to cryptographic mechanisms that approximate random functions rather than random permutations. A random permutation won't ever suffer an internal collision with itself when a different input value has been used.

(or k-tuples) of long keys, an attacker may be able to mount a kind of related-key attack on our key-stretching scheme. If long keys are highly nonrandomly distributed, it may even be possible to guess a significant subset of the possible long keys without any stretching computations.

3.4 Full and Partial Precomputations

In ordinary keysearch attacks, if an attacker can force the use of some chosen plaintext, or can count on the use of some known plaintext in almost all cases, he can carry out a single precomputation of all possible keys, and index them by the expected ciphertext block from the chosen or known plaintext.

In attacking a key-stretching algorithm, a slightly better alternative is available. An attacker can carry out the full 2^{s+t} trial encryptions and rekeyings once, and wind up with a list of 2^s possible long keys. Each message can now be attacked using an s-bit search. (Note that this can even be done on massively parallel hardware under some circumstances, e.g. by sorting the long key list, and preloading a different subset of the long keys to each parallel search machine.)

Similarly, an attacker can carry out a partial precomputation, computing some subset of 2^{s-u} possible long keys, to get a 2^{s-u} search per message, with a 2^{-u} probability of getting the correct key in any message.

This is an inherent problem with key-stretching algorithms: they all are slightly more susceptible to precomputation in this way. The best that any key-stretching algorithm can do is not to be any more vulnerable than this to precomputation or partial precomputation attacks. In practice, it is more practical to do a $s+y$-effective-keylength precomputation attack, with a chosen- or known-plaintext that will appear in almost all messages the attacker wishes to break. One goal of key stretching is that we can set t large enough that an $s+t$-bit search is no longer practical. Otherwise, we will need to use a salt value as described in Section 2.1.

4 Key Stretching Algorithms

4.1 A Hash-Function-Based Key Stretching Algorithm

Let $H()$ be a secure, collision-resistant one-way hash function, such as SHA1 [NIST93] or RIPE-MD160 [DBP96], and let S be a salt value.

The basic scheme works as follows:

$X_0 = H(K_{short}, S)$.
For $i = 1$ to 2^t:
  $X_i = H(X_{i-1})$.
$K_{long} = X_{2^t}$.

We are computing the 2^t-th iterate of $H()$. We will denote the ith iterate of $H()$ as $H^i()$. Hence,

$$K_{long} = H^{2^t}(K_{short}, S).$$

Considering each of our design criteria in turn, we can make the following statements of security:

Time to Compute: One crucial requirement is that naive iteration must be the fastest way to compute $H^{2^t}(x)$ for random inputs x. In other words, there must be no shortcuts available to implement $H^{2^t}()$. Fortunately, we have some theoretical results which assure us that such shortcuts can't exist, if H has sufficient resistance to collision attacks.

We say that an m-bit hash function $H()$ is strongly collision-resistant if the best way to find an unknown collision is with a naïve birthday attack. Normally this requires at least $2^{m/2}$ work (where the unit of time is how long it takes to compute $H(x)$ for a randomly chosen m-bit string x). We must be careful, though: a permutation fits this definition, yet our proof of security depends on the ability to find collisions using a naïve birthday attack. Hence we add the additional technical requirement that a naïve birthday attack is likely to find a collision for $H()$. We expect that a truly random function would admit a birthday attack, so if $H()$ is a pseudorandom function then we also expect that it will admit a birthday attack. In other words, this second requirement should be satisified in practice for most reasonable hash functions; it is primarily technical in nature.

Given these two properties of $H()$, we have the following result.

Theorem 1. *If $H()$ is a strongly collision-resistant hash function that admits a naïve birthday attack, then computing the i-th iterate of H requires time at least $i/2$.*

We leave the proof for the appendix.

This theorem shows that one cannot really compute the 2^t-th iterate of $H()$ any faster than by straightforward iteration, as the ability to find a shortcut would imply a weakness in the underlying hash function. In short, we get very close to the amount of strength from key stretching that one would expect intuitively: the theorem says that iterating the hash function 2^t times adds at least $t-1$ bits of extra "strength" to the key.

In fact, our proof yields an even stronger result, which is applicable even in the case that there is a way to slightly speed up collision attacks on the hash function. We prove that if finding a collision takes at least $2^{m/2-c}$ time, for some $c > 0$, then iterating the hash function 2^t times adds at least $t-c-1$ bits of "strength." Note that for some hash functions which have slight weaknesses against collision search, it may be possible to truncate the hash and achieve even better proofs of security, though there are some limits to when this is possible.

We should emphasize that the theorem assumes that the input X is chosen at random. Hence, it shows that the *average-case* time-complexity for computing the stretched key is close to 2^t.

Cherry Picking: Unfortunately we are not able to show that the *best-case* time-complexity is also 2^t. In fact, given X and $H^{2^t}(X)$, one can compute $H^{2^t}(Y)$ for a different Y in time 1. Simply let $Y = H(X)$, for then

$H^{2^t}(Y) = H(H^{2^t}(X))$. Therefore it is difficult for us to make a statement about the security of this scheme against cherry picking. Because of the redundancy in the input to the hash function, we believe it will be secure given a secure hash function, but cannot prove it. This issue is related to the reuse of internal values, discussed below.

Non-Random Long Keys: There should not be any significant number of non-random long keys. By assumption, $H()$ is a strongly collision-resistant hash function. If one could predict $H(Y)$ without knowing Y, with better than random chance, then one could produce collisions for $H()$ faster than with a naïve birthday attack.

Reusing Internal Values: If the output of our hash function is large enough, then we are unlikely to produce duplicate values X, X' along with i and j $(1 \leq i, j \leq 2^t)$ such that $H^i(X) = H^j(X')$ where $X \neq X'$ or $i \neq j$. Hence we expect that an attacker will not be able to reuse values in order to speed up a brute force search.

4.2 A Block-Cipher-Based Key Stretching Algorithm

Let $E_K(X)$ be the encryption under some block cipher, E, of X under key K. In this case E should have a key length and block length which are each at least K_{long} bits long. Then, this method is:

$K_0 = (K_{short}, S)$.
For $i = 1$ to 2^t:
  $K_i = E_{K_{i-1}}(0)$.
$K_{long} = K_{2^t}$.

For ease of notation, let $f(k) = E_k(0)$. Then

$$K_{long} = f^{2^t}(K_{short}||S)$$

Remark: *Note that we run into some practical problems with this scheme when we want to stretch a key to more bits than the block size of the cipher. We can easily get around this by using two independent blocks in our computation, though this makes it harder to prove the security of the resulting key-stretching scheme. Alternatively, we may try to find a block cipher of appropriate size.*

Considering each of our design criteria in turn, we can make the following statements of security:

Time to Compute: Unfortunately we are not able to make as strong a statement for this approach as for the hash-based approach. One observation we have made is that the ability to compute $f^{2^t}()$ in time faster than 2^t would speed up a time/memory trade-off attack [H80]. It will not be a significant speed-up, but a speed-up nonetheless.
If we can prove that $f^1()$ is a strongly-collision resistant hash, then we can apply the analysis from the previous section. However, there is no guarantee that this property will hold.

Cherry Picking: As with the hash function approach, it is difficult to make assertions about the strength of this cipher against cherry picking. We expect that a set of keys for which computing $Stretch_{E,2^t}()$ can be done in time significantly less than 2^t would also point out a weakness in the underlying cipher. However, we are not able to make a formal proof to this effect.

Non-Random Long Keys: Suppose it is possible to learn K_{i+1} without knowing K_i. Therefore, it is possible to know $E_K(X)$ given X but not K. This is a major weakness in the block cipher. In fact, if we can reliably guess $E_K(X)$ without knowledge of any u bits of K, then we have just found a way to reduce the effective keyspace of E by u bits.

Reusing Internal Values: A secure block cipher can be viewed as a pseudorandom permutation family. Therefore one would expect that $E_{k_1}(0) = E_{k_2}(0)$ with at most slightly better than random chance (otherwise this would provide a method for distinguishing the permutation family from a random permutation family contradicting its pseudorandomness). So provided that $s + t$ is sufficiently small compared to the block length, we expect that there will not be any repeated internal values.

5 Making Massively Parallel Hardware More Expensive

Just as with ordinary keysearch attacks, one concern we have with attacks against a key-stretching algorithm is whether or not it is feasible for an attacker to build a special-purpose massively-parallel keysearch machine. For example, we may have a secure key-stretching scheme which stretches 40-bit keys to an effective strength of 56 bits. Unfortunately, it is feasible for a moderately well-funded attacker to build a machine to carry out a 56-bit keysearch of some cipher like DES [Wie94, BDRSSTW96]. We would like our key-stretching algorithms to be hard implement on very cheap hardware, the kind that would most likely be used in building a keysearch machine.

In making cheap hardware implementations of our scheme more difficult, it is important to keep our ultimate goal in mind: We want to raise the cost of keysearch machines to the point where it is too expensive to build. However, we must remember not to do this at the cost of usability of the key-stretching scheme. If a user must choose to stretch her 40-bit key to only 56-bits, instead of 64, because of the added expense of our hardware-frustrating techniques, then we've most likely made the attacker's job easier. Typically, we will be interested in hardware-frustrating techniques that don't cost more than one or two extra bits of stretching, but make the parallel search engines significantly more expensive to build.

5.1 Modifying the Hash-Based Approach

In Section 4.1 we proposed a scheme using hash functions. Most hash functions proposed—SHA-1, MD5, etc.—are designed for efficient implementations. One approach is to use hash functions which do not yield efficient implementations;

however, people are not likely to design such hash functions. Instead we can modify existing hash functions to yield expensive hash functions, without voiding their security warranty.

Let $f()$ be a one-way, strongly collision-resistant hash function with an m-bit output and $E()$ an m-bit permutation. Then we define a new hash function F as

$$F(X) = E(f(X)).$$

It is easy to show that $F()$ is also a one-way, strongly collision-resistant hash function. Suppose that $O(Y)$ was an oracle which could invert $F()$, then we can turn this into an oracle $O'(Y)$ for inverting $f()$ by setting $O'(Y) = O(E(Y))$. Similarly, if $F(X) = F(X')$ for some $X \neq X'$, then $f(X) = f(X')$. Hence it is no easier to find collisions for $F()$ than it is to find collisions for $f()$.

So we can make an expensive hash function $F()$ by making $E()$ an expensive permutation. Using the resulting hash function $F()$ we can apply the analysis of Section 4.1.

Note that in order for these arguments to be valid, $E()$ must not depend on the input to $F()$.

5.2 A Design for a Computationally Expensive Permutation

Our permutation $E()$ needs to involve operations that are expensive to provide on cheap hardware, but not on a general-purpose processor. Two obvious places to look for this are 32-bit arithmetic and use of moderately large amounts of RAM.

Let C0,M0 and C1,M1 be parameters for a linear congruential generator, such that the generator never overflows a 32-bit register. (In other words, $(M0-1) * C0 + 1 < 2^{32}$ and $(M1-1) * C1 + 2 < 2^{32}$.) Let $T[0..255]$ be a 256-element table of 32-bit unsigned values.

One might build the permutation using an involution (up to replacement of f by $-f$):

$$t = w[0] + w[1] + w[2] + w[3] + w[4]$$

$$w[0] = w[0] + f(0,t)$$

$$w[1] = w[1] + f(1,t)$$

$$w[2] = w[2] + f(2,t)$$

$$w[3] = w[3] + f(3,t)$$

$$w[4] = w[4] - f(0,t) - f(1,t) - f(2,t) - f(3,t)$$

This has the effect of altering every bit of the hash output with a minimum number of operations. The whole permutation might be computed as follows:

```
temp = word[0]+word[1]+word[2]+word[3]+word[4];
A0 = temp % M0;
A1 = temp % M1;
u = temp;
for(i=0;i<256;i++){
A0 = (A0 * C0 + 1) % M0;
A1 = (A1 * C1 + 1) % M1;
T[i] = A0^A1^u;
u = rol(u,13)+T[i];
}
j = 255;
sum = 0;
for(i=0;i<256;i++){
u = rol(u,6)^T[j];
word[i%4] += u;
sum += u;
j = (j+T[j])%256;
}
word[4] -= sum;
```

Note that this is intended as a discussion of a possible way to do this. We have not spent enough time researching this algorithm to make any strong claims about it, other than that it is reversible. However, it is useful to remember that, because we have hash functions on both sides of it, it doesn't need too much cryptographic strength.

The basic purpose of this permutation is to require the availability of 256 32-bit words of RAM and 32-bit arithmetic operations in order to carry out a key-stretching.

6 Related Work

Some work on key stretching for block ciphers was independently done by [QDD86]. Also, our techniques are similar in spirit to the technique of using repeated iterations of DES to make UNIX password-guessing attacks more expensive [MT79].

Other authors have also independently examined the problem of hardening password-protection schemes based on one-way functions. Manber proposed one simple approach based on hashing a random "salt" with the password; the resulting hash digest is stored, but the salt is securely deleted [Man96]. Abadi, Lomas, and Needham recently independently proposed a scheme for strengthening passwords which is very similar to Manber's [Aba97]; Abadi et. al. also show how to apply that approach to communication security in a setting similar to the one we envision for key stretching. They point out several important differences between key stretching and password strengthening, which are worth repeating:

- Strengthening requires extra work from only one party; key stretching adds to the workload of both endpoints to the communication. In some applications, these computational savings could be a critical factor. Also, if the endpoints have access to multiple CPUs, strengthening can be easily parallelized, whereas stretching a key is an inherently sequential process.
- Strengthening actually adds information-theoretic entropy, while stretching does not, so with key stretching thorough use of salts is especially important to prevent dictionary-style attacks.
- The two approaches seem to rely on somewhat different properties of the underlying hash function. Local one-wayness seems to be critical for password strengthening. In contrast, the analysis of Section 4 suggests that key stretching may rely most heavily on the hash function's resistance to collision attacks.

  The collision-resistance properties of today's cryptographic hash functions are arguably somewhat better-analyzed than their local one-wayness properties. On the other hand, our proofs make extremely strong assumptions of collision-resistance, so our theoretical analysis may not be as strong as we'd like for practical use.

Finally, there has been much work on using public-key techniques to strengthen cryptosystems that must rely on low-entropy shared secrets. However, these require heavy-weight public-key algorithms and complex protocols. In contrast, one of our central design requirements was that our key stretching schemes should not require implementation of any new cryptographic primitives: if both endpoints already have a shared block cipher, where possible we should re-use that cipher to stretch keys, or if a hash function is already available, it should be used, but no new primitives should be introduced. In short, key stretching provides a simpler and lighter-weight version of the public-key approaches; the power of key stretching is not as great as that achievable with public-key cryptography, but it is still substantial.

7 Summary, Conclusions, and Further Directions

In this paper, we have introduced the concept of *key-stretching*: a method of increasing the difficulty of trying all possible values for some low-entropy variable by a fixed, well-understood amount. We have also given constructions for two key-stretching schemes, one based on a block cipher and the other on a hash function. The hash-based construction was shown to have the property that a method for speeding up the exhaustive search of the low-entropy variable translates into a significant weakness in the underlying hash function.

Of course, key stretching is no substitute for long keys. Key stretching only reduces a short key's resistance to certain attacks. If at all possible, system designers should use long keys.

Several open questions remain. It would be nice to see an analysis of various key schedules' economies of scale for keysearch attacks. For example, Peter

Trei [Tre97] has recently demonstrated that a software DES keysearch can step through the DES key schedule in Gray code order, making each new key much cheaper to schedule for the keysearcher than a key is to schedule for an ordinary user. What other ciphers have this or worse properties, which make keysearch attacks easier to carry out? What design principles can a cipher designer follow to ensure that there aren't economies-of-scale for keysearch attacks on his cipher? What design principles exist for designing permutation algorithms that will require certain hardware support to compute efficiently?

8 Acknowledgments

The authors would like to thank Matt Blaze and Martín Abadi for their helpful comments.

References

[Aba97] M. Abadi, personal communication.

[BDRSSTW96] M. Blaze, W. Diffie, R. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Wiener, "Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security," January 1996.

[DBP96] H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160: A Strengthened Version of RIPEMD," *Fast Software Encryption: Third International Workshop, Cambrdige, UK, February 1996 Proceedings,* Springer-Verlag, 1996, pp. 71–82.

[H80] M.E. Hellman, "A Cryptanalytic Time-Memory Trade Off," *IEEE Transactions on Information Theory*, v. 26, n. 4, Jul 1980, pp. 401–406.

[Knu81] D. Knuth, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms,* Addison-Wesley, 1981.

[Man96] U. Manber, "A Simple Scheme to Make Passwords Based on One-Way Functions Much Harder to Crack," *Computers & Security*, v. 15, n. 2, 1996, pp. 171–176.

[MT79] R.H. Morris and K. Thompson, "UNIX Password Security," *Communications of the ACM*, v. 22, n. 11, Nov 1979.

[NBS77] National Bureau of Standards, NBS FIPS PUB 46, "Data Encryption Standard," National Bureau of Standards, U.S. Department of Commerce, Jan 1977.

[NIST93] National Institute of Standards and Technology, NIST FIPS PUB 180, "Secure Hash Standard," U.S. Department of Commerce, May 93.

[QDD86] J.-J. Quisquater, Y. Desmedt, and M. Davio, "The Importance of 'Good' Key Schemes (How to Make a Secure DES with $\leq$ 48 Bit Keys?)," *Advances in Cryptology—CRYPTO '85 Proceedings*, Springer-Verlag, 1986, pp. 537-542.

[Sch96] B. Schneier, *Applied Cryptography, Second Edition*, John Wiley & Sons, 1996.

[Tre97] Peter Trei, *personal communication*, 1997.

[Wie94] M. Wiener, "Efficient DES Key Search," TR-244, School of Computer Science, Carleton Unversity, May 1994.

A Theoretical analysis of hash iteration

We give the proof of Theorem 1 here. First, we must prove the following lemma:

Lemma 2. *Suppose we can evaluate the 2^m-th iterate of a b-bit hash function h in $2^m/l$ time on average, for some $m < b/4$. Then we can find a collision in h about $l/2$ times as fast as a naive birthday attack.*

Proof: Let $k = 2^{b/2}$. Define $f(j) = h^j(a)$ for some fixed starting point a. Suppose that computing the 2^m-th iterate h^{2^m} of h can be done in $2^m/l$ work, for some $l > 1$. We know $2^m < \sqrt{k}$, so assume $l < \sqrt{k}$. We show how to find a collision in h with $2k/l$ work.

Note first that computing $f(i+j)$ from $f(i)$ takes only j/l work, if $j \geq 2^m$. Compute $f(i+2^m)$ from $f(i)$ with $2^m/l$ work, then compute $f(i+2\cdot 2^m)$ from that with $2^m/l$ work, and so on, until you reach $f(i+j)$; you'll need $j/2^m$ steps, and each step takes $2^m/l$ time, so the total time needed is j/l.

The typical method of finding a collision is as follows: by the birthday paradox, the sequence $f(0)$, $f(1)$, $f(2)$, …, $f(k)$ will hit a cycle with very high probability. This leads immediately to a collision in h. (If i, j are the least $i < j$ with $f(i) = f(j)$, then $f(i-1)$ and $f(j-1)$ are two colliding preimages of h.)

In our optimized attack, we compute the following values:

$$f(k), f(k+\sqrt{k}), f(k+2\sqrt{k}), \ldots, f(2k), f(2k+1), f(2k+2), \ldots, f(2k+\sqrt{k}).$$

Now we claim that this optimized sequence has two powerful properties. First, if the original sequence $f(0), \ldots, f(k)$ has a collision, then the optimized sequence will too. Moreover, the optimized sequence can be enumerated significantly more efficiently than the original sequence.

First we establish that if the original sequence $f(0), \ldots, f(k)$ cycles and gives a collision, then the optimized sequence will too. Note that the optimized sequence can be viewed as an application of the baby-step giant-step method. The second half of the sequence, namely the values $f(2k), \ldots, f(2k+\sqrt{k})$ form a sort of "landing pad": if the original sequence enters a cycle of period less than k, then one of the values $f(k + j\sqrt{k}), j = 0, \ldots, \sqrt{k}-1$ will hit the landing pad and cause a collision in the optimized sequence. (Why? Let p be the period of the cycle. If $p \leq \sqrt{k}$, then the second half of the optimized sequence will have a collision. Otherwise, $\sqrt{k} < p < k$, and there must be some i, j with $0 \leq i, j < \sqrt{k}$ such that $i = j\sqrt{k} \bmod p$. The latter statement follows just because it is impossible to have $\sqrt{k}$ mutually disjoint subsets of values modulo p, if each subset has cardinality $\sqrt{k}$ and if $k > p$.)

We see that the optimized sequence yields a collision whenever the original sequence does. Furthermore, finding collisions in the optimized sequence is easy, when they exist. For example, it suffices to use a hash table with $2\sqrt{k}$

entries, or to sort the $2\sqrt{k}$ values; the time required for setting up the data structure will be small compared to complexity of computing the sequence.

Next we examine how long it takes to compute the optimized sequence. Computing $f(k)$ can be done in k/l time from $f(0)$. (See above.) Also, $f(k+\sqrt{k})$ can be computed in $\sqrt{k}/l$ time from $f(k)$, and $f(k+2\sqrt{k})$ can be gotten in another $\sqrt{k}/l$ time, and so on. Therefore, computing the first half of the optimized sequence requires $2k/l$ work. Also, we can compute $f(2k+1), f(2k+2), \ldots, f(2k+\sqrt{k})$ from $f(2k)$ with $\sqrt{k}$ time.

Therefore, the total time to compute the optimized sequence is $2k/l + \sqrt{k}$. The latter term will be negligible, so for simplicity we call it $2k/l$.

This means that we can find a collision in $2k/l$ time by taking advantage of the oracle for evaluating the 2^m-th iterate of h. A naive birthday attack would take time $2^{b/2} = k$, so our optimized technique is $l/2$ times faster than that.

QED.

Corollary 3. *Suppose you're given a b-bit hash function h where finding a collision takes at least $2^{b/2-c}$ time, for some c. Suppose further that a naïve birthday attack works against this hash. Then you can't evaluate h^{2^m} in less than 2^{m-c-1} time on average, and key-stretching with h^{2^m} adds at least $m-c-1$ bits of "strength" to the key.*

The theorem then follows directly from the corollary.

Theorem 4 1. *If $H()$ is a strongly collision-resistant hash function that admits a naïve birthday attack, then computing the i-th iterate of H requires time at least $i/2$.*