One Way Hash Functions and DES

Ralph C. Merkle
Xerox PARC
3333 Coyote Hill Rd.
Palo Alto, CA. 94304
merkle@xerox.com

ABSTRACT

One way hash functions are a major tool in cryptography. DES is the best known and most widely used encryption function in the commercial world today. Generating a one-way hash function which is secure if DES is a "good" block cipher would therefore be useful. We show three such functions which are secure if DES is a good random block cipher.

Introduction

DES can be used to build a one-way hash function which is secure if DES is a "good" block cipher. Previous efforts have not produced a satisfactory result[4,5,11,12] (although recent work at IBM[18, 19] appears very hopeful). We will use the meta-method discussed by Merkle[1,2] for the construction of one-way hash functions. Similar ideas have also been presented by Damgaard[20] and Naor and Yung[3].

One way hash functions (also called MDC's (Manipulation Detection Codes), fingerprints, cryptographically secure checksums, one way functions, and others) have been generally known for some time. The first definition was apparently given by Merkle [1,2] who also gave a method of constructing one-way hash functions from random block ciphers. More recent overviews have been given by Jueneman, Matyas, and Meyer[11], Jueneman[4], and Damgaard[20]. The method of construction used by Merkle was provably secure[2], provided that the block cipher was "random" (see below). Naor and Yung[3] also proved security results in the context of polynomial time reducibilities.

We will review the definitions and method of construction required to build a one-way hash function. We will then show how DES can be used to build a function that satisfies the desired properties, assuming that DES is "random." Because DES is

G. Brassard (Ed.): Advances in Cryptology - CRYPTO '89, LNCS 435, pp. 428-446, 1990.

in fact non-random in several known special cases, the constructions shown here would have to be modified to take this into account before being used in a real system.

Definitions

Broadly speaking, there are two definitions for one-way hash functions. The first definition is for a "weak" one-way hash function. A weak one-way hash function is a function F such that:

1.) F can be applied to any argument of any size. For notational convenience, F applied to more than one argument is equivalent to F applied to the bit-wise concatenation of all its arguments.

2.) F produces a fixed size output. (The output might be 56 bits).

3.) Given F and x, it is easy to compute F(x).

4.) Given F and a "suitably chosen" (e.g., random) x, it is computationally infeasible to find an x' $\neq$ x such that F(x) = F(x').

The phrase "computationally infeasible" used above can be replaced with any of several more precise definitions -- we defer a more detailed definition for purposes of this paper till later, but note in passing that each precise definition of computational infeasibility will in turn result in a somewhat different definition of a one way hash function.

In a weak one way hash function there is no guarantee that it is computationally infeasible to find pairs of inputs that map onto the same output. That is, it might be that F(z) = F(z') for some inputs z and z' that someone in fact found. However, if x $\neq$ z and x $\neq$ z', then this doesn't matter. Clearly, we must prevent someone from deliberately picking x so that it is equal to a value of z or z' which they know has this undesirable property. What's more, it must be difficult to generate too many z-z' pairs (and it clearly must be impossible to generate z-z' pairs in which we can effectively control the value of z or z') or even a randomly chosen x would not be safe. Thus, choosing x in a random fashion should make it unlikely that anyone can find an x' such that F(x)=F(x'). It is possible, however, to choose x in a non-random fashion and still be safe. If we assume that F is random (often a reasonable assumption in practice as discussed later), then any method of choosing x that does not depend on F is effectively random with respect to F and therefore suitable. This observation is sometimes important in practice, for it allows simpler (therefore faster and cheaper) methods of selecting x. Weak one-way hash functions have been described based on DES[12].

Various methods of randomizing x have been proposed. Merkle[2] proposed that

x be encrypted with a good block cipher using a truly random key. The key would be pre-pended to the resulting ciphertext, and the new message would then be random. Damgaard[private communication] proposed selecting a short random pre-fix to x. When hashed in the manner suggested here and in [20], such a random pre-fix would effectively randomize the entire hash. Naor and Yung[3] do not randomize x, instead they randomly choose the function applied to x from a family of functions (also providing the advantages of parameterization discussed in the next paragraph). These various methods have different advantages and disadvantages, depending on the specific objectives being pursued.

Weak one-way hash functions also suffer from the problem that repeated use weakens them. That is, if a weak one-way hash function is used to hash 1,000 different random values for x (which might represent 1,000 different messages signed by a 1,000 different people) then the chances of finding an x' such that *one* of the thousand values of x satisfies $F(x) = F(x')$ might be 1,000 times higher. As more people sign more messages with the same weak one-way hash function, the overall security of the system is degraded. This undesirable state of affairs can be dealt with by parameterizing the family of one-way hash functions F. We simply define a family of one-way hash functions with the property that each member F^i of the family is different from all other members of the family, and so any information about how to break F^i will provide no help in breaking F^j for $i \neq j$. If the system is designed so that every use of a weak one-way hash function is parameterized by a different parameter, then overall system security can be kept high.

The less mnemonic term "property 2" was used in [2] -- the term "weak" seems preferable.

The alternative definition is of a "strong" one-way hash function. A strong one-way hash function is a function F such that:

1.) F can be applied to any argument of any size. For notational convenience, F applied to more than one argument is equivalent to F applied to the bit-wise concatenation of all its arguments.

2.) F produces a fixed size output. (The output might be 112 bits).

3.) Given F and x, it is easy to compute F(x).

4.) Given F, it is computationally infeasible to find any pair x, x' such that $x \neq x'$ and $F(x) = F(x')$.

Strong one-way hash functions are easier to use in systems design than weak one-way hash functions because there are no pre-conditions on the selection of x, and they provide the full claimed level of security even when used repeatedly (or misused either because of error or sub-optimal system design). Many authors recomend the exclusive use of strong one-way hash functions[4,11] because of the practical

difficulties of insuring that the pre-conditions on x imposed by a weak one-way hash function are met, and the increased problems in insuring that different parameters are selected for each use. In practice, the constraints imposed by use of a weak one-way hash function imply that x cannot be chosen by an agent who has a motive to break the system. If A signs a message which B provides, then A will have to "randomize" the message by some means before signing it. If A fails to randomize the message, then B can select a "rigged" message, and later surprise A by exhibiting a novel message and showing that A signed it.

We shall not consider weak one-way hash functions further, except to note they can be quite valuable in some system designs, although they must be used with caution. Parameterized weak one-way hash functions in particular appear to be under-rated. Although the design process is more complex, systems built in this fashion will require half as much storage for the "y" values. In essence, this is because the size of the output for a strong one-way hash function must be twice as large as one might expect to avoid "birthday" or "square root" attacks (see later). That is, a typical weak one-way hash function might have an output size of 56 bits, and might require 2^{56} operations to break. A typical strong one-way hash function might have an output size of 112 bits but still require only 2^{56} operations to break.

The phrases "it is computationally infeasible" and "it is easy to compute" can be replaced with many different definitions. A fully satisfactory definition for practical applications must consider both constant factors and average case complexity. DES has been criticized for having a key size that is "too small" (only 56 bits), while a slightly larger key size (64 bits) would eliminate most criticism. A satisfactory definition of "computationally infeasible" should be able to distinguish meaningfully between these two cases. For this reason, the definition we use here is stronger than the often-used definitions based on polynomial time reductions. This stronger definition allows us to prove stronger results (results of more practical cryptographic significance) but in exchange we must provide better informal reasons for believing that a function reasonably approximates our definition.

For purposes of this paper we will use the "random function", "demon in a box" or "oracle" model of complexity for DES. That is, we will assume that DES is actually random, in the sense that encryption and decryption are accomplished by looking up the correct value in a large table of random numbers. The table is secret and information about table entries can be learned only by requesting the value for some table entry -- which costs one "operation". More or less equivalently, we can view DES as a black box with a demon inside -- whenever we wish to encrypt or decrypt a value with some key, we must slip the input into the box and ask the demon to tell us the value. The demon will generate a truly random number and return it to us, again charging us one "operation" for his efforts. The demon promises to be consistent -- once he's given you the ciphertext for a given key and plaintext, he will always return the same ciphertext when given the same key and plaintext. He will also give you the correct plaintext if you happen to give him the ciphertext and key

just mentioned.

Although we assume that DES is random, it is not immediately obvious how to apply DES to a large x and produce a small output in a secure manner. Several previous efforts have failed[4,5,11,13]. (A recent proposal by IBM[18, 19] looks very hopeful and has been carefully scrutinized within IBM. Although there is as yet no proof that it is correct, should such a proof be found the IBM proposal would then be preferable because of its superior performance). The successful attacks on these methods do not require any knowledge of the internal structure of DES and so would still work if DES were truly random. Therefore, it is useful to show a method of generating a one-way hash function by using repeated applications of DES which is immune from this class of attacks.

While there has been a lively debate about the actual security level provided by DES[14,15,16,17] the fact remains that no major deviations from random behavior have been found. The most significant deviation noted would effectively reduce the key size by one bit to 55 bits [9]. To circumvent the problem caused by this non-random behavior would require the redesign of the proposals made here to use a 55-bit block cipher, with a corresponding slight reduction in either performance or security. Other known cases of non-randomness in DES would require case-by-case analysis and possible redesign. Aside from the few known cases of non-random behavior, DES is generally viewed as a "random" function in the literature[17] and it seems almost certain that *some* block cipher exists with the desired properties, regardless of the eventual fate of DES. The existence of "random block ciphers" seems almost assured in some intuitive sense, and DES appears to be a reasonable approximation to one.

The primary use of one-way hash functions is authentication[1,11]. If $y = F(x)$, and we are already confident that we know y, then we can obtain x from any source (in particular, an untrustworthy, unreliable but cheap and convenient source) and verify that $y = F(x)$. Because it is computationally infeasible to find an x' such that $y = F(x')$, we now have confidence that we have the correct value of x.

The advantages of using one-way functions are clear if, for example, x is a one-megabyte document, while y is 112 bits. If we can design a system that provides good security for y, then we can always double-check x any time we wish -- thus, we can be much more casual (i.e., spend less money) in storing x. One-way hash functions are virtually a necessity in digitally signing large messages because most digital signature methods are rather inefficient. If we first reduce a one-megabyte x to a small y, then signing the small y is much easier. One-way hash functions are a universal building block in authentication systems, and have been used to define practical digital signatures[1,2,6] and to detect fraudulent changes in messages[4,11]. When used in a tree-structure, one-way hash functions can be used to conveniently authenticate any individual entry in the "public directory" (so often needed in public-key based systems)[1,7].

The meta-method

We first define a meta-method (given in [1,2]) for constructing F. In this meta-method, we build F (which accepts an arbitrarily large argument, x) from a simpler function F_0. The definition of F_0 is exactly the same as the definition of F, except that F_0 accepts only a fixed size argument. That is, F_0 might accept an input of 224 bits, while producing a smaller output of perhaps 112 bits. As we shall see, the larger the input that F_0 accepts the better, for this will mean we can more rapidly hash down the arbitrarily large input, x. We require, however, that F_0 have an input that is larger than its output.

We can now define F in terms of F_0 as follows:

```
Function F(x) returns FixedSizeOutputString; -- This might be 112 bits
x: ARRAY[1 .. n] OF convenientlySizedChunks; -- perhaps 112 bits
-- Note that SizeOf(input to F0) =
-- SizeOf(convenientlySizedChunks) + SizeOf(FixedSizeOutputString).
result: FixedSizeOutputString; -- perhaps 112 bits
BEGIN
result = 0;
FOR i = 1 to n DO result = F0(result,x[i]);
RETURN(result);
END;
```

As an aside, we note that x is padded with 0's until its size is an integral multiple of the size of the convenientlySizedChunks. Note that padding with 0's might introduce some ambiguity about the exact value of x that is being authenticated, as discussed by Jueneman[4]. The message "010110" padded to 8 bits would be "01011000" and it is now unclear how many trailing 0's were present in the original message. Several methods are available to resolve this ambiguity. We select a method that will also be useful in resolving a further problem: we append the length of x, in bits, at the end of x. To make this additional "length" field easy to find, we shall right justify it in the final block. If the length field won't fit, we add additional blocks to the end of x. For purposes of notation, we shall assume that the final few blocks of x hold this count; for example, if we desire a 64-bit count and the "convenientlySizedChunks" are 8 bits, then the count would occupy x[n-7], x[n-6], x[n-5], ... x[n-1], and x[n].

As a second aside, the linear pattern of computing F used above can be replaced with a tree pattern, as discussed in [1, 2, 7 page 170]. This tree structure is very useful when we wish to authenticate not the entire input, but only a single item (or leaf) from the input. This occurs in practice when, for example, we wish to authenticate the public key of a single user from the public directory. Certainly, if we know the correct hash value for the entire public directory, we could re-hash and re-verify the whole thing before believing that the single entry we were interested in was correct. This, however, is very inefficient. If the one-way hash function used a tree pattern, then we could authenticate a single leaf node (the one entry in the public directory that we"re interested in) if we knew only that entry, and the log n entries along the authentication path from the leaf to the root. This reduces O(n) computations to verify the whole public directory to O(log n) computations to verify the single leaf entry we"re interested in.

In looking at the definition of F we note that the input, x, is treated as an array of "conveniently SizedChunks". These chunks will have a size which is the size of the input to F_0 less the size of the output from F_0. If the input is 224 bits, and the output is 112 bits, then each chunk will be 224-112 = 112 bits. Clearly, bigger chunks imply fewer applications of F_0, which will make the computation of F more efficient. On the other hand, the chunks can be as small as one bit[3,20], which is inefficient but still secure.

Using this construction, we can now prove that breaking F (finding an x and x' such that $F(x) = F(x')$ and $x \neq x'$) is at least as hard as breaking F_0. This is easily shown by induction. We first make the assumption that x and x' have the same length -- we shall relax this assumption later.

Basis:

for $n=1$, $y = F(x) \equiv F_0(\mathbf{0}, x[1])$ (where **0** is the all zero bit pattern).

Clearly, breaking F must be as hard as breaking F_0 in this case, for F and F_0 are the same. That is, if $x \neq x'$ and $F(x) = F(x')$ and $n=1$, then $x[1] \neq x'[1]$ and $F_0(\mathbf{0}, x[1]) = F_0(\mathbf{0}, x'[1])$. Thus, by definition a "break" of F implies a "break" of F_0.

Induction:

We know the property holds for n, and we wish to show it holds for $n+1$.

$y = F(x) \equiv F_0(F(x[1 .. n]), x[n+1])$.

Neither $F(x[1 .. n])$ nor $x[n+1]$ (i.e., neither of the arguments to F_0) can be modified successfully, for then we would have broken F_0 directly. But if $F(x[1 .. n])$ is correct then, by the induction hypothesis, x[1 .. n] cannot have been modified without breaking F_0. This means that no bit of x could have been modified without breaking F_0. Q.E.D.

In the more general case, it might be that the length of x is not the same as the length of x'. We can assume, without loss of generality, that x is shorter than x'. (If this is not the case, then simply swap the names of x and x'). If x is shorter than x', and F(x) = F(x'), then the proof given above shows that x is a postfix of x'. In particular, the length fields of both x and x' must be the same. It is now a simple matter to reject the string whose length is incorrect.

The particular method selected here to solve the problems caused when x is a pre-fix or postfix of x' is only one of many. Several other techniques are possible. Which method is actually used will vary depending on the precise circumstances and design objectives.

The ability to prove that a break of F implies a break of F_0 works for most definitions of complexity. This will be useful later in the paper when we construct an F_0' and an F_0" that are not random, but such that breaking them is still "computationally infeasible".

As an aside, it should be noted that a random block cipher in which the size of the key was greater than the desired input size to F_0 could easily be used to generate a suitable F_0. We need only define $F_0(x)$ as ENCRYPT(x, 0) (where x is used as the key, and 0 is the plaintext input consisting of all 0's). If the key size were, say, 224 bits then our problem would be well solved. That was exactly the approach taken in [1,2]. Construction of random block ciphers with large keys seems well within current capabilities. DES, for example, uses 768 bits of key material internally (16*48 bits). Any problems in using DES as a one-way hash function stem almost exclusively from its limited key size. This paper primarily addresses the specific problems involved in using DES as currently defined as the basis for a one-way hash function -- if DES had a significantly larger key size then the problem solved here would be trivial.

With the construction of F from F_0 now in hand, it is clear what to do: define F_0 in terms of DES. This is simpler than trying to define F in terms of DES directly -- which has been tried and can lead to subtle problems[4,5,11]. Defining F_0 in terms of DES will involve a fixed number of applications of DES in some particular pattern. Analyzing the complexity of a fixed pattern of applications of DES can be tedious, but has proven easier than analyzing the indefinite number of applications of DES that are required when F is defined directly in terms of DES.

A Method

In our first method, we shall simply attempt to produce an F_0 such that its input is larger than its output; we won't worry about efficiency. In our second and third methods, we'll see improvements in efficiency but at the cost of more complex

analyses. The methods presented here are by no means unique, nor is there any reason to believe they are the most efficient. Further work should produce improved methods. The principle, however, should be clear -- try to combine a few applications of DES in such a way that we produce a satisfactory F_0 -- (and such that the analysis is tractable...).

For this first method we will adopt a fixed size output of 112 bits for both F and F_0 because we are building a strong one-way hash function which we wish to be as secure as DES, i.e., it must require at least 2^{56} operations to break. The output size must be at least 112 bits because we know that smaller values of the output will make the system vulnerable to "square root attacks"[4,5,8,11]. If the output size were 56 bits, then hashing 2^{28} arbitrarily chosen messages would result in a high probability that two of those messages produced the same 56-bit output. Because the probability of a collision goes up sharply as the number of messages exceeds the square root of the number of possible output values, such attacks are called "square root attacks" (also known as "birthday attacks"[5] because the probability that two people at a party were born on the same day goes up sharply when the number of people at the party exceeds the square root of 365, or about 19 people).

We turn now to some definitions. DES accepts 64 bits of input, 56 bits of key material, and produces a 64-bit output. We denote encryption by:

64-bit-ciphertext = ENCRYPT(56-bit-key, 64-bit-plaintext)

We denote decryption by:

64-bit-plaintext = DECRYPT(56-bit-key, 64-bit-ciphertext)

We can view ENCRYPT and DECRYPT as functions that map from 120 bits onto 64 bits. That is, ENCRYPT and DECRYPT can be viewed as very large tables which have random (or nearly random -- see below) entries. We do not initially know the random values in these tables, and can find out only by using one of the two functions ENCRYPT or DECRYPT. (The concept of random encryption functions was given by Shannon[10]). In some sense, our 2^{120} random 64-bit entries are just 2^{126} random bits. These 2^{126} random bits can be used in any way want -- if we desire a large number of random bits, we need only look up many different values in our giant table. We are fundamentally limited only by the total number of random bits, not the particular format they are packaged in. We can repackage the bits to best suit our needs -- which is exactly what we will be doing.

Unfortunately, ENCRYPT (and similarly DECRYPT) is non-random in the following sense: we can easily generate a legitimate input-output pair in which the output value is non-random. Even worse, we can easily generate an input-output pair in which the output value is anything that we want! Given a 64-bit value that we want produced as the output, we can easily find many 120-bit inputs that produce the desired output value. That is, we can easily find many keys and plaintexts such that ENCRYPT(key,plaintext) = ciphertext. We simply pick a random 56-bit key, and

then decrypt the 64-bit ciphertext with that key. This is bad. To remedy this, we can XOR the 64-bit output with the 64-bit input, to produce a new 64-bit output. We will define this operation with a lower case f_0 (this definition has been used before[12]). By definition, if the 120-bit input to f_0 is arbitrarily divided into a 56-bit key and a 64-bit plaintext, then

f_0(key,plaintext) ≡ ENCRYPT(key, plaintext) XOR plaintext

All our further use of DES will be through this new function, f_0. This function maps 120 bits onto 64 bits. It will be a good approximation to a random function if DES is a good approximation to a random function. In particular, f_0 has the interesting property that *the output value will be random no matter how it is computed.* If we compute n values of f_0 by computing DES n different times, then the n output values of f_0 will be randomly distributed. In contrast, the input values might be quite systematic -- we could simply compute $f_0(0)$, $f_0(1)$, $f_0(2)$, ... $f_0(n-1)$. The sequence of input values 0, 1, 2, ... n-1 certainly qualifies as non-random. However, no matter how hard we try, the output values will be random, as can be easily proven. First, we note that there are only two possible methods of computing an output value of f_0 -- either we encrypt some plaintext with DES, or we decrypt some ciphertext. If we encrypt some plaintext, then the ciphertext produced by the encryption is random (courtesy of our demon). When we exclusive-or this random ciphertext with the (possibly non-random) plaintext, the result is also random. Equivalently, if we decrypt a ciphertext, then the plaintext produced by the decryption is random (again courtesy of our demon). When we exclusive-or this random plaintext with the (possibly non-random) ciphertext, the result is also random. As a result, the output of f_0 is always random, no matter how it was computed.

In this paragraph we first discuss and then decide we can safely ignore a minor deviation from randomness caused by the fact that DES is a permutation. If we compute ENCRYPT(key, plain1)=cipher1, then we know that computing ENCRYPT(key, plain2)=cipher2 will *not* produce cipher1, i.e., that cipher1 ≠ cipher2. (We are simply observing that each plaintext has one and only one ciphertext, and each ciphertext has one and only one plaintext. For this reason, DES is not quite random). This, however, is a very minor deviation from randomness in the context in which we are using DES. Recall that we only wish to achieve a level of security equivalent to DES, which means that we only need to insure that f_0 behaves randomly for at most 2^{56} different values applied to the input. Now, if we have actually applied f_0 2^{56} times, then at worst there will not be 2^{64} possible output values left, but instead only $2^{64} - 2^{56}$. This corresponds to about 63.994 bits -- a loss which we can and will neglect. (Put another way, this means that our demon might compute a random 64-bit value and then occasionally reject the value because it's already been used for some other plaintext-ciphertext pair. The demon will reject a random value at most one time in 256. This slight deviation from randomness has almost no practical impact on the proofs that follow).

Because f_0 is such a good approximation to a random function, we will be unable

to find two inputs which map onto a 64-bit output in significantly less than 2^{32} operations (as a consequence of the birthday problem). This, however, does not provide good enough security. We want to force cryptanalysis to take at least 2^{56} operations, which will require an output of 112 bits. To accomplish this, we will simply look up "x" twice, and concatenate the two outputs. This will produce 128 bits. This is more than we need, so we throw away the extra bits. However, we only have a single function f_0 -- how can we look up x twice? By reducing the size of x from 120 bits to 119 bits, and using the additional bit to effectively split f_0 into two different functions.

Formally, we declare that x is 119 bits. We define F_0 as:

$F_0(x) \equiv$ First112bitsOf(f_0("0", x), f_0("1", x))

$F_0(x)$ is simply the concatenation of the two applications of f_0. We first prefixed x with a "0", and then with a "1" to distinguish the two applications of f_0. To produce the desired 112 bit output, we threw away the final 16 bits of the 128 bits produced.

Intuitively, f_0 is just a very large random table. The index into this table is a 120 bit number. By making x only 119 bits in size, we effectively produce two tables -- the first half of f_0 and the second half of f_0. By looking up x first in the first half, and then in the second half, we obtain two totally unrelated random numbers. This produces 128 random bits from a single value of x. Now, we need only throw away 16 bits. We are left with 112 random bits, which is what we desired.

As a result, we have a random function F_0 which accepts a 119 bit input and produces a 112 bit output. Therefore we can use F_0 to build a one-way hash function F that will accept 119-112 = 7 bits per iteration. Each iteration requires two computations of DES, so we require one application of DES for every 3.5 bits to be hashed. The performance is poor -- but we can prove rather easily that it's as secure as DES under the assumption that DES is a random function.

A Faster Method

We can show a faster method is also secure, though the analysis is somewhat more complex. The faster method will require that we divide x into two pieces: x_1 of 118 bits and x_2 of 54 (= 120-64-2) bits. In total, x will be 172 (118+54) bits. We will reduce these 172 bits to 128 bits using 4 applications of DES, which allows us to hash (172-128)/4 = 11 bits per application of DES. This is clearly better than 3.5, though better is still possible.

We will define F_0' as follows:

F_0' $\equiv$

a: f_0("00", c: f_0("10", x_1), x_2),

b: f_0("01", **d**: f_0("11", x_1), x_2)

Note that F_0' (and so F') produce a 128 bit output. We will use this additional output only to guarantee 56 bits of equivalent security -- we will not provide the full 64 bits of security that might seem possible, but will instead "waste" a few bits to make the construction go through. (The actual security is somewhat better than 56 bits, though we will not prove this).

Notationally, we have labeled the intermediate values in the computation with the letters **a**, **b**, **c**, and **d**.

The degradation in security in this computation occurs because different values of x_1 might produce the same intermediate values for **c** or **d**. That is, it might happen that **c** = f_0("10", x_1) = **c**" = f_0("10", x_1"). If such collisions did not occur, then we could guarantee that different values of x_1 would produce different values for both **c** and **d**. This in turn would let us guarantee that all pairs x, x' such that $x \neq x'$ would result in selection of the values for both **a** and **b** from different entries in our giant table. That is, if $x \neq x'$ then either $x_2 \neq x_2$" or both (**c** ≠ **c**" and **d** ≠ **d**"). But then the two values used as input to f_0 to compute **a** and **a**" must be different, and the two values used as input to f_0 to compute **b** and **b**" must also be different. Thus, we have guaranteed that **a** and **b** were selected from different locations in our giant table, which lets us conclude that the probability of a collision in both **a** and **b** is a random event whose probability we can compute, and which is small (less than one in 2^{56}).

Of course, collisions involving **c** or **d** *will* occur, and so the foregoing logic is false. However, we can bound the number of collisions that are expected to occur, and use this bound to determine a bound on the deterioration in the security of F_0'.

We observe that **c** and **d** are always random, because they are produced as outputs from f_0 and we have already shown that outputs from f_0 are always random, no matter how computed. Therefore, no matter how cleverly x_1 is chosen, the probability that the same value of **c** (or of **d**) is produced by two different values of x_1 is random. Therefore, if we limit ourselves to 2^{56} applications of f_0, the expected maximum number of collisions will be $2^{56} * (2^{56}/2^{64})$ or 2^{48}. (This also holds for computations of values of **d**). Given that there are at most 2^{48} such collisions, the expected maximum number of 3 way collisions is $2^{56} * (2^{48}/2^{64}) = 2^{40}$. The expected maximum number of n-way collisions is 2^{64-n*8}, which implies there are probably no 8-way collisions. We can use a 7-way collision as an upper bound.

Now, if $F_0'(x) = F_0'(x')$, and $x \neq x'$, then either $x_1 \neq x_1$" or $x_2 \neq x_2$". If $x_2 \neq x_2$", then the computations of **a** and **b** were random, and the probability of a random collision for both of them is negligible (if 2^{56} computations of **a** and **b** had already been done, the probability of a collision would be $2^{56}/2^{128}$ -- negligible as far as we"re concerned). If, on the other hand, $x_1 \neq x_1$" and $x_2 = x_2$", we can further divide the situation into two cases: either (**c** = **c**" or **d** = **d**"), or (**c** ≠ **c**" and **d** ≠ **d**"). If (**c** ≠ **c**" and **d** ≠ **d**") then by the logic used before the computations of **a** and **b** were random, and the probability of a random collision for both of them is negligible. If, on the

other hand, (**c** = **c**" or **d** = **d**") then we can use our upper bound on the number of collisions to limit the number of distinct values of x_1 for which this can occur. (We reject the case where **c** = **c**" and **d** = **d**" as being sufficiently improbable that we can ignore it). We can assume that **c** = **c**" (symetrical considerations hold if, instead, **d** = **d**"). Obviously there are at most 7 different values for x_1 that map onto the same value for **c**. Therefore, instead of getting random values when we pick **a** and **b**, we might at worst get 7 non-random trials in which the 7 values computed for **a** were always the same (because **c** and x_2 were always the same, and hence **a** = f_0("00", **c**, x_2) would be the same). This would generate 7 different trials for **b**, but **b** is only 64 bits -- so these 7 trials have a higher probability of success than 7 trials that randomly selected a 128-bit value. Specifically, the probability of success in these 7 random trials for **b** is at most $7^2/2^{64}$ or $49/2^{64}$. This produces an equivalent security level of (64 - $\log_2$ 49) or 58.3 bits. This is greater than 56, as desired. (This does not imply our security level is 58.3 bits -- remember that we have already assumed a limit of 2^{56} in a few places. It simply confirms that we can reach at least 56 bits of security. For various reasons, we could actually achieve more than 56 bits of security -- but this suffices to show the idea).

Although the foregoing discussion was relatively informal because of the relative simplicity of the problem, the techniques become harder to apply in more complex cases. In the following case, a more formal analysis was needed because of the sheer complexity of the situation.

A Complex And Yet Faster Method

It is again possible to improve the performance, though to do so requires a significantly more complex analysis. We will divide a 234 bit x into 2 pieces, each of 117 bits in size: x_1 and x_2. We will define F_0" as:

F_0" ≡

f_0("00", First59bitsOf(f_0("100", x_1)), First59bitsOf(f_0("101", x_2))),

f_0("01", First59bitsOf(f_0("110", x_1)), First59bitsOf(f_0("111", x_2)))

Essentially, we have built a small tree of f_0's. Because there are six applications of f_0, we have used the first two or three bits to divide f_0 into six distinct functions. Thus, f_0("00", ...), f_0("01", ...), f_0("100", ...), f_0("101", ...), f_0("110", ...) and f_0("111", ...) can be viewed as six unrelated random functions. The "leaf" functions in this tree map 117 bits onto 59 bits. The "root" functions map 118 bits onto 64 bits. Overall, F_0" maps 234 bits onto 128 bits using six applications of DES, which means we can hash 234-128 = 106 bits/iteration, or 106/6 = almost 18 bits/application of DES. This is an improvement over 11 bits/application of DES.

However, we are now left with the problem of showing that we have not significantly degraded security. Again, there *is* degradation in security caused by collisions during the intermediate computations. This is why we kept 128 bits of output -- F_0" is not perfectly random and we must retain additional output bits in order to reach our desired objective of 2^{56} operations to break it. Our proof will actually not be able to show that we have retained the full 56-bits of security that we desire, though we will come close. Tightening up the proof would make it even more complex, though probably (although not certainly) providing the 56-bit security level desired.

First, we shall label the various values produced by this computation.

F_0" ≡

a: f_0("00", **c**: First59bitsOf(f_0("100", x_1)), **e**: First59bitsOf(f_0("101", x_2))),

b: f_0("01", **d**: First59bitsOf(f_0("110", x_1)), **f**: First59bitsOf(f_0("111", x_2)))

We first note that **c**, **e**, **d**, and **f** are random. No matter how cleverly we pick x_1 and x_2, these values are random and we can apply statistical methods to them. In particular, every value of x_1 and every value of x_2 will generate a tuple; that is, every value for x_1 will generate a tuple <**c**,**d**>, and every value for x_2 will generate a tuple <**e**,**f**>. Because f_0 is random, the actual values of x_1 and of x_2 are irrelevant -- the only thing that matters is that we have generated tuples <**c**,**d**> and <**e**,**f**>.

This leads to our first **definition**: we define a ***random linkage map*** as two sets of tuples, each tuple having two chosen 59-bit elements, and each set of tuples containing 2^{56} elements. The two 59-bit elements in a tuple are "linked" because they are generated from a single value of x_1 or of x_2. Note that this is equivalent to saying that the two elements in a tuple are generated randomly, hence the name *random* linkage map.

Intuitively, a random linkage map is all the useful information that any algorithm can ever hope to obtain about the four possible intermediate values. A random linkage map actually requires 2^{58}th computations of f_0 to compute, so it actually is an upper bound on the information that can be obtained. Any actual algorithm that attempts to crack F_0" will in fact have less information than is present in a random linkage map. However, it can't hurt to give the algorithm additional information for free. Any optimal algorithm to crack F_0" should not slow down if it is given all the information in a random linkage map, instead of getting only the sub-set of information about a random linkage map that it actually computed. The major reason for providing all the information in a random linkage map is that, although an actual algorithm would use only part of the information, it is not clear which part it would select. By providing all the information, we avoid the problems involved in determining optimal strategies for dealing with the partial information that can actually be computed.

For any computation of F_0" and any actual value of x, we will generate four intermediate values **c**, **e**, **d**, and **f**. By definition, both <**c**,**d**>, and <**e**,**f**> will appear in the corresponding random linkage map. This motivates the following **definition**: a quadruple <**c**,**e**,**d**,**f**> is ***doubly linked*** with respect to a random linkage map if the tuple <**c**,**d**> appears in the first set and the tuple <**e**,**f**> appears in the second set.

Given a doubly linked quadruple <**c**,**e**,**d**,**f**>, we can compute a valid output of F_0". This output is valid because there exist an input x_1 concatenated with x_2, where x_1 links the tuple <**c**,**d**> and x_2 links the tuple <**e**,**f**>, which generates the intermediate quadruple <**c**,**e**,**d**,**f**> from which the output is then computed. That is, a doubly linked quadruple is just as good as an actual input, x.

We now prove that the expected running time for an optimal algorithm to find two values x, x' such that F_0"(x) = F_0"(x') (where F_0" is based on DES in the manner described, and DES is assumed to be random), is at least as long as the expected running time for an algorithm to find a pair of doubly linked quadruples such that both quadruples generate the same output (the same values for **a** and **c**), given only a random linkage map and the ability to compute values for **a** and **c** using f_0. That is, the algorithm that uses the random linkage map cannot use f_0 to compute new values for **c**, **e**, **d**, and **f** (after all, it already has the random linkage map which is supposed to provide at least as much information as could ever be obtained by computing such intermediate values with f_0 -- so letting it compute more intermediate values would provide it with an unfair advantage). Instead, valid intermediate quadruples must be obtained from the linkage map. Both algorithms can apply f_0 to arguments that are prefixed with "0", for these are just the values used to compute either **a** or **c**. The random linkage map contains no information about computations of **a** or **c**.

The proof is relatively simple -- given the foregoing definitions. If we are given any algorithm for cracking F_0" we can use it to define an algorithm that is just as good at solving an equivalent problem defined in terms of the random linkage map. Given a random linkage map and an F_0" cracking algorithm, we run the F_0" cracking algorithm, but now lie to it whenever it tries to compute f_0. Instead of giving it the "correct" truly random value, we instead give it a value selected at random from the random linkage map. Of course, we must be consistent. If the optimal algorithm gives us the same argument twice, we return the same value. In addition, if it gives us a value for which we"ve already returned a **c**, and now requests a **d**, we must return the proper linked value. This does not introduce any bias, though, because all the entries in a random linkage map are random. The truly random values generated by the DES "oracle" are just as good as the truly random values taken from the random linkage map, and so the expected running time of this "random linkage map" cracking algorithm must be less than or equal to the expected running time for the corresponding F_0" cracking algorithm. Therefore, a lower bound on the expected running time of a "random linkage map" cracking algorithm is also a lower bound on the expected running time of any F_0" cracking algorithm.

We can now concentrate on finding a lower bound for the running time of an

algorithm to crack a random linkage map. We start by analyzing the intermediate values **c**, **e**, **d**, and **f** in a random linkage map.

We define a ***collision*** for **c**, **e**, **d**, or **f** with respect to a linkage map as a value of **c**, **e**, **d**, or **f** which appears in two different tuples in the same set and in the same position in the two tuples.

First, because **c**, **e**, **d**, and **f** are 59 bits, an upper bound on the expected number of collisions for each of them is $(2^{56}/2^{59}) * 2^{56}$ or 2^{53}. The number of triple collisions will be bounded by $(2^{53}/2^{59}) * 2^{56}$ or 2^{50}. In general, the expected number of n-tuple collisions is bounded by 2^{59-3*n}. There will probably not be any 20-tuple collisions, so we can safely use this as the expected maximum.

Our objective is to find a lower bound for the running time of the best algorithm that finds two doubly linked quadruples <**c**,**e**,**d**,**f**> and <**c**",**e**",**d**",**f**"> such that they produce the same output, i.e., such that **a** ≡ f_0("00", **c**, **e**) is equal to **a**" ≡ f_0("00", **c**", **e**"), and **b** ≡ f_0("01", **d**, **f**) is equal to **b**" ≡ f_0("01", **d**", **f**"). (Note that the two quadruples are each doubly linked internally, independently of the other -- the links do not extend from one quadruple to the other). To do this, we will consider what happens during an actual run of a linkage-map cracking algorithm. The only things that such a run can do are compute a value of **a** from some tuple <**c**,**e**>, or compute a value **b** from some tuple <**d**,**f**> (note that these tuples are not the tuples that appear in the linkage map -- those were tuples <**c**,**d**> and<**e**,**f**>). We can ask, each time such a computation is performed, what the probability is that that particular computation will result in finding a pair of quadruples with the desired property -- i.e., the probability that that particular computation of f_0 will terminate the run successfully. Clearly, if we can provide an upper bound on the probability of success for each such computation during the course of the run, then we can determine a lower bound on the expected running time.

Now, if we were to compute **a** = f_0("00", **c**, **e**) then we could succeed if and only if there were already two doubly linked quadruples <**c**,**e**,**d**,**f**> and <**c**",**e**",**d**",**f**">, and further the case that **b** = f_0("01", **d**, **f**) = f_0("01", **d**", **f**"). We would then succeed if, after computing **a**, we found that it matched the value for **a**", i.e., **a** = f_0("00", **c**, **e**) = **a**" = f_0("00", **c**", **e**"). If there were only one other quadruple <**c**",**e**",**d**",**f**"> such that **b** = **b**", then the probability of success would be one chance in 2^{64}. However, there might be several. In particular, it might be the case that **b** = **b**" because **d** = **d**" and **f** = **f**". The only other alternative is that **b** = **b**" and either **d** ≠ **d**" or **f** ≠ **f**". In the first case, because there are at most 20 collisions for either **d** or **f**, there could be at most 20^2 = 400 quadruples matching this criteria. Obtaining a bound for the second case is more difficult, but is possible by noting that every distinct computation of **a** produces a random number.

There are at most 2^{56} computations of **b**. Therefore, the expected maximum number of collisions for **b** is 2^{48}. The number of triple collisions is 2^{40}, the number of quadruple collisions is 2^{32}, and the number of n-way collisions is 2^{64-8*n}. Clearly, the

expected number of 8-way collisions is 0, and can be neglected. The maximum number of 7-way collisions is 2^8, which is rather small. We could use 7 as a simple bound on the number of collisions, or we could perform a more complex analysis to show that, on average, the number of collisions is more like 2 or 3. If we content ourselves with the easier bound of 7, we can then produce a bound on the probability of success following each computation of **a** or **b**.

For each quadruple <**c**,**e**,**d**,**f**> there are at most 400 quadruples <**c**",**e**",**d**",**f**"> such that **b** = **b**" because **d** = **d**" and **f** = **f**". Further, there are at most 7 collisions such that **b** = **b**" and either **d** ≠ **d**" or **f** ≠ **f**". Therefore, there are at most 7*400 = 2,800 quadruples <**c**",**e**",**d**",**f**"> which might cause the computation of **a** to terminate the run. Therefore, the probability of success is upper bounded by $2{,}800/2^{64}$. This corresponds to 64 - $\log_2$ 2,800 bits of security, or 64 - 11.5 or about 52.5 bits. This is somewhat lower than we desire (by 3.5 bits) but it seems likely that tightening the proof would recover most if not all of this loss.

The most obvious places where this lower bound could be tightened are the following. First, we always assumed 2^{56} operations could be performed for the computation of all intermediate values. Clearly, this is not possible. In fact, these 2^{56} operations need to be parcelled out among all computations of all values in some optimal way. Second, we gave away a great deal of information for free. This information would in fact have to be computed by some means. Third, we used simple bounds of 7 collisions for elements **a** or **b** and 20 collisions for elements **c**, **e**, **d**, or **f**. These upper bounds are achieved only infrequently. That is, if the upper bound of 20 were achieved for only a hundred elements, then it would only improve the overall probability of success modestly. The more frequently occuring values of 10 or 11 would have greater significance, for they would involve the bulk of the computations actually made.

Conclusion

We have shown three methods for building a strong one-way hash function from DES. All methods are provably secure if DES is a random function. All methods rely on producing a "building block" function which is of fixed and finite size, and using this "building block" to build the actual one-way hash function which can then accept an input of indefinite size. In the first method, a simple pattern of two applications of DES was used, and the proof was not complex. The resulting method, though, was not very efficient. The second method improved the efficiency, but a moderately complex analysis of a particular pattern of four applications of DES was used to prove the required security properties. The final method improved efficiency further, but a complex analysis of six applications of DES was used to prove that the security level was at least equivalent to 52.5 bits -- and areas where the proof could be "tightened up" (hopefully to the desired 56-bit level of security) were noted. There is no reason

to believe that this particular pattern of six applications of DES is optimal -- indeed, it would be very surprising if it were. It seems probable, therefore, that more efficient patterns of application of DES exist, and can be derived using the general methods outlined here.

Acknowledgements

The author would like to thank many people for their interest and comments, and would particularly like to thank Don Coppersmith, Ivan Damgaard, Dan Greene, Mike Matyas, Carl Meyer, and Moti Yung.

Bibliography

1.) "Secrecy, Authentication, and Public Key Systems", Stanford Ph.D. thesis, 1979, by Ralph C. Merkle.

2.) "A Certified Digital Signature", unpublished paper, 1979. To appear in Crypto '89.

3.) "Universal One-Way Hash Functions and their Cryptographic Applications", Moni Naor and Moti Yung, Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing, Seattle, Washington May 15-17, 1989, page 33-43.

4.) "A High Speed Manipulation Detection Code", by Robert R. Jueneman, Advances in Cryptology - CRYPTO '86, Springer Verlag, Lecture Notes on Computer Science, Vol. 263, page 327 to 346.

5.) "Another Birthday Attack" by Don Coppersmith, Advances in Cryptology - CRYPTO '85, Springer Verlag, Lecture Notes on Computer Science, Vol. 218, pages 14 to 17.

6.) "A digital signature based on a conventional encryption function", by Ralph C. Merkle, Advances in Cryptology CRYPTO 87, Springer Verlag, Lecture Notes on Computer Science, Vol. 293, page 369-378.

7.) "Cryptography and Data Security", by Dorothy E. R. Denning, Addison-Wesley 1982, page 170.

8.) "On the security of multiple encryption", by Ralph C. Merkle, CACM Vol. 24 No. 7, July 1981 pages 465 to 467.

9.) "Results of an initial attempt to cryptanalyze the NBS Data Encryption Standard", by Martin Hellman et. al., Information Systems lab. report SEL 76-042, Stanford University 1976.

10.) "Communication Theory of Secrecy Systems", by C. E. Shannon, Bell Sys. Tech. Jour. 28 (Oct. 1949) 656-715

11.) "Message Authentication" by R. R. Jueneman, S. M. Matyas, C. H. Meyer, IEEE Communications Magazine, Vol. 23, No. 9, September 1985 pages 29-40.

12.) "Generating strong one-way functions with cryptographic algorithm", by S. M. Matyas, C. H. Meyer, and J. Oseas, IBM Technical Disclosure Bulletin, Vol. 27, No. 10A, March 1985 pages 5658-5659

13.) "Analysis of Jueneman's MDC Scheme", by Don Coppersmith, preliminary version June 9, 1988. Analysis of the system presented in [4].

14.) "The Data Encryption Standard: Past and Future" by M.E. Smid and D.K. Branstad, Proc. of the IEEE, Vol 76 No. 5 pp 550-559, May 1988

15.) "Defending Secrets, Sharing Data: New Locks and Keys for Electronic Information", U.S. Congress, Office of Technology Assessment, OTA-CIT-310, U.S. Government Printing Office, October 1987

16.) "Exhaustive cryptanalysis of the NBS data encryption standard", Computer, June 1977, pages 74-78

17.) "Cryptograhy: a new dimension in data security", by Carl H. Meyer and Stephen M. Matyas , Wiley 1982.

18.) "Secure program code with modification detection code", by Carl H. Meyer and Michael Schilling; Proceedings of the 5th Worldwide Congress on Computers and Communication Security and Protection -- SECURICOM 88, pp. 111-130, SEDEP, 8, Rud de la Michodiese, 75002, Paris, France.

19.) "Cryptography -- A State of the Art Review," by Carl H. Meyer, COMEURO 89, Hamburg, May 8-12, 1989, Proceedings - VLSI and Computer Peripherals, 3rd Annual European Computer Conference, pp. 150-154.

20.) "Design Principles for Hash Functions" by Ivan Damgaard, Crypto '89.