On The Security of Key Derivation Functions

Carlisle Adams[1], Guenther Kramer[2], Serge Mister[2], and Robert Zuccherato[2]

[1] University of Ottawa, Ottawa, Ontario, Canada K1N 6N5
cadams@site.uottawa.ca
[2] Entrust, Inc., 1000 Innovation Drive, Ottawa, Ontario, Canada K2K 3E7
{guenther.kramer, serge.mister, robert.zuccherato}@entrust.com

Abstract. Key derivation functions are commonly used within many cryptographic schemes in order to distribute the entropy contained in an uneven way in a long stream of bits into a string that can be used directly as a symmetric key or as a seed for a pseudo-random number generator, or to convert short strings such as passwords into symmetric keys. This paper examines the common key derivation function constructions and shows that most of these have some concerning properties. In some situations, the use of these key derivation functions may actually limit the security that would otherwise be obtained. A new construction is also provided which seems to have better properties and an intuitive justification for its security is given.

1 Introduction

Hash functions are used to map arbitrarily long sequences of bits into a small, fixed number of bits. They are used as a building block in many fundamental cryptographic schemes. For example, they are used for generating a message representative to be digitally signed, for distributing the entropy contained in an uneven way in a long stream of bits (e.g., a Diffie-Hellman shared secret, or the bits collected from various user and environmental inputs on a computer) into a (possibly shorter) string that can be used directly as a seed for a pseudo-random number generator or as a symmetric key, and for converting short strings such as passwords into symmetric keys.

As computers have gotten faster, the size of keys used in cryptographic applications has increased. For example, AES [8] has key sizes of 128, 192 and 256 bits, compared with DES [6], which had an effective key size of 56 bits. For most applications, 128 or 160 bits of security is almost certainly sufficient. The amount of computing power required to mount an attack on this level of security is well beyond anything currently available. However, the development of AES and the process that was used to determine Rijndael as the AES cipher, have made it acceptable to discuss "attacks" that reduce the security of ciphers, even if such "attacks" are drastically infeasible. In addition, a number of documents, including the National Institute of Standards and Technology's Key Management Guidelines document [16], provide guidance on how to obtain "equivalent" cryptographic strength between various types of algorithms that

K. Zhang and Y. Zheng (Eds.): ISC 2004, LNCS 3225, pp. 134–145, 2004.

may be used in an application and recommend that applications only use algorithms with "equivalent" strength. Governments and businesses that rely upon cryptography are thus expecting and requiring advanced levels of cryptographic protection, whether or not that protection is actually needed from the point of view of a strict cryptographic risk analysis.

Therefore, many applications require 256 bits of keying material for use with a block cipher, and 256 bits (or more) of entropy to seed a random number generator that will generate the keys. Hence, a straightforward use of many current hash functions will generate fewer bits than required for the application. This has led to the development of constructions, often based on a counter (for example, [2,12,13,15]), for the last two applications above that make repeated use of a hash function to generate a larger output. This paper analyses those constructions (commonly referred to as *key derivation functions*) in light of the details of existing hash implementations. Properties of the constructions are highlighted that limit their field of use. In particular it is shown that many currently used constructions are insufficient for their intended purpose when a stipulated goal of the system is to use only algorithms of "equivalent" strength. Finally a new construction is proposed that appears to overcome the highlighted limitations.

Despite the necessity of using key derivation functions in most, if not all, cryptographic protocols, there has been surprisingly little work done on analyzing the security that they provide. Some related research was presented in [4], but the focus of that work was to construct variable-length input pseudorandom functions (VI-PRFs) from fixed-length input pseudorandom functions (FI-PRFs) such as DES [6] or the compression functions of MD5 [19] and SHA-1 [7]. The motivating applications for that work are message authentication and encryption. For the latter, the authors mention the need for a variable-length output pseudorandom function, but do not pursue the construction of such VO-PRFs. In any case, by its very definition, the cascade construction does not transfer the entropy in a large input string to a large output string (since the output is a fixed, small size), which is precisely the characteristic we desire for a good key derivation function. A different kind of construction is therefore required.

Thus, with this paper we hope that we can stimulate further research in this needed area. The results of this paper are concerning not simply because they exist, but because they have existed for so long and been incorporated into so many protocols without any serious discussion of the security properties desired and provided by the constructions used. More research is required in order to fully understand the security level provided by these key derivation functions.

2 Hash Functions and Key Derivation Functions

The results of this paper apply to the use of any of the current commonly used hash functions in a key derivation function. In particular, they apply to any hash function construction that uses a compression function to mix the current block of data with a chaining variable that is the output of the application of the compression function on the previous block. We will use SHA-1 [7] as a concrete

example in the remainder of this paper. It takes as input any bit string of length up to 2^{64} bits, and outputs a fixed-length 160 bit output. However, we note that the techniques described in this paper apply to all hash functions that take the same general form as SHA-1. The important point is that the hash function takes blocks of size B and combines each successive block with an internal chaining variable, H, of size *hashlen*, to produce a hash value also of size *hashlen*.

2.1 Overview of Key Derivation Functions

Let us assume that we have a string S, of length L, which is produced from a source with e bits of entropy, $e \leq L$. The string S has some property such that we do not want to use it directly as a symmetric key or as a seed for a random number generator. Perhaps S is a master key from which other keys must be derived, or perhaps it has some structure that does not make it suitable for direct use, or perhaps it is not the correct length. Thus we want a process that will produce a derived key k of length l with $\min(e, l)$ bits of entropy. This means that it should require $O\left(2^{\min(e,l)}\right)$ work for an attacker that does not know S, or is unable to determine any partial information about k, to determine k. In other words, we desire a key derivation function such that whatever entropy is present in the input is evenly distributed over the output.

This is a reasonable and desirable property for a good key derivation function to have. Intuitively, it says that partial knowledge of k gives an adversary no significant advantage in determining any unknown bits in k.

We note that this property depends upon the source used for the input. For example, given a key derivation function, kdf, that produces strings of length 128 bits from inputs of length 256 bits, we could define a source that generates only those 256 bit inputs x such that $kdf(x)$ is the all 1's string. On average there will be around 2^{128} such inputs. Now, the function, when considered over all possible 256 bit inputs, may have our desired property. However, when considered over our modified input source, it certainly does not.

Thus, it is crucial to know the probability distribution P_S of the source that produces S and the interaction between the key derivation function and the source. It seems reasonable to assume however that most "natural" sources of random strings will not have such negative interactions with reasonably constructed key derivation functions, and that is what we have assumed in our constructions. We believe a necessary condition for a secure key derivation function is that the above property hold for all but a negligible number of sources for S.

We also note that the related problem of converting a string S from an arbitrary source with entropy e into a uniformly distributed string of entropy l has a provable solution that makes use of universal hash functions (see [11]). However, we desire a construction that does not use universal hash functions, but rather uses something with an efficiency similar to existing key derivation functions and uses any cryptographically secure hash function.

Let us now consider the performance of the following common construction of a key derivation function with respect to our desired property. Let $h()$ be any

hash function of the form described in Section 2.1. In particular, $h()$ could be SHA-1. Let $n = \lfloor \frac{l-1}{hashlen} \rfloor + 1$ and let $[x]_l$ be the leftmost l bits of x. The symbol $||$ represents concatenation and $h(x, y) = h(x||y)$. Also let P_i be a known (e.g., standardized) string containing certain defined parameters of the key derivation function. These parameters must have the property that $P_i \neq P_j$ if $i \neq j$. A common way to accomplish this is for P_i to contain the value i. Let $kdf(S, l) = [h(S, P_1)||h(S, P_2)|| \cdots ||h(S, P_n)]_l$.

Now, assume that the length of S is a multiple of the hash function block length, *hashlen*. Thus, when computing $h(S, P_i)$, the string S will end on a block boundary. Now, consider the chaining variable H after processing just the string S. It has length *hashlen* but is the only state within the hash function that depends upon S. Since the attacker knows the parameter string P_i, she can exhaust over all $2^{hashlen}$ values of H and determine all possible values of $h(S, P_i)$. Thus, if an attacker is given any particular $h(S, P_i)$ she can determine the value of H after just the string S was processed (which is the same regardless of which parameters will be processed next) with $2^{hashlen}$ work, by exhausting over all values of H and producing candidates for $h(S, P_i)$ for each value of H. Once the correct value of H is determined, she can use that to determine $h(S, P_j)$ for $j \neq i$, and hence determine the remaining bits of the output of $kdf(S, l)$. Therefore, this key derivation function does not distribute the input entropy evenly over the output string if $l > hashlen$ and $e > hashlen$ since knowledge of *hashlen* bits of k, along with $2^{hashlen}$ work, is sufficient to reveal all remaining bits of k, regardless of the length of k. Notice that this could commonly be the case in systems that wish to generate keys for AES-256 [8], but support only the SHA-1 hash function (i.e., 2^{160} work reveals the full 256-bit key).

We note that this attack on the defined key derivation function does not appear to be new. It seems to be known or at least hinted at in various discussion groups and mail lists, but we have been unable to find a formal reference for it. Also, the above, general, flawed construction has repeatedly been used in national and international standards that require a key derivation function. See Section 4 for more details.

Now let us consider a slightly different key derivation function. With the same terminology and components as before, let us now define $kdf(S, l) = [h(P_1, S)||h(P_2, S)|| \cdots ||h(P_n, S)]_l$.

It appears that the attacker cannot now use the fact that S remains fixed for all applications of $h(P_i, S)$ since the chaining variable H depends upon P_i and will be different each time that the hash function begins processing S. However, let us now give our attacker the ability to influence the parameters P_i. This is not unreasonable since in many cases the parameters will contain identifiers for the parties that will have access to the derived key, identifiers for the applications or protocols in which the keys will be used, or other data that can be manipulated. Assume also that the length of P_i is a multiple of the hash function block length.

Now by the birthday paradox the attacker, with $2^{\frac{hashlen}{2}}$ work, can find two parameter strings P_i and P_j, $i \neq j$, such that the internal chaining variable H is the same after processing these parameter strings [20]. Since S remains fixed and is the only data processed by the hash function after this point, we get that

$h(P_i, S) = h(P_j, S)$. Thus, given $h(P_i, S)$ the attacker will know $h(P_j, S)$ and this construction does not have our desired property against an attacker that can influence the parameters, if $l > hashlen$ and $e > hashlen$.

What is the problem with these constructions and why are strong hash functions not providing us with the characteristics that we want? The answer to these questions lies in the underlying approach to the construction. Creating a *kdf* by varying the input to the hash function in a known way (i.e., by appending or prepending known parameter blocks) relies upon the property that knowing some portion of the input to a hash function reveals no information about its output. But cryptographic hash functions have exactly the opposite property: they are designed to be one way and collision resistant. That is, given the *output* it is hard to determine any information about the *input*.

Existing cryptographic hash functions were not designed to have the "partially-known input, unknown output" property. They were never designed to mix any particular bits of the input into the internal variables of the hash in such a way that the output could not be determined without the knowledge of those bits. Furthermore, it seems somewhat naïve to hope that these hash functions will have such a property by accident. Thus, if we wish to use existing hash functions as a building block for a key derivation function, we must design the *kdf* to rely upon only the properties that hash functions were designed to have. In particular, we must design the *kdf* to use the one wayness and collision resistance of cryptographic hash functions.

We note in passing that designing key derivation functions by modifying the input to a hash function seems like a very ad-hoc construction. What is required is a general construction that can be shown to be secure based only upon the designed properties of the underlying hash function. Then any hash function that has those properties can be used in this construction and the designer can be sure that the desired properties of the key derivation function have been obtained.

While the work factors given above (i.e., $2^{hashlen}$ and $2^{\frac{hashlen}{2}}$) will likely be acceptable security levels for some environments with some choices of hash function (e.g., SHA-512 [7]), the fact that existing, common constructions do not distribute the input entropy evenly over the output string is concerning. We would recommend that standards and applications transition to a key derivation function that provides more appropriate security guarantees as soon as it is feasible to do so.

2.2 Key Derivation Functions Using HMAC

The HMAC [14] construction is defined as $HMAC(k, data) = h(k \oplus opad || h(k \oplus ipad || data))$ where $h()$ is a cryptographic hash function (such as SHA-1), k is a secret key, and *ipad* and *opad* are inner and outer fixed strings, respectively, that are used to mask k.

Resulting from the definitions of *ipad* and *opad*, the strings $k \oplus ipad$ and $k \oplus opad$ each end on a block boundary for $h()$. Therefore, an attacker can, with $2^{hashlen}$ work, try all possible chaining variables that result from $h(k \oplus ipad)$ and

also, with $2^{hashlen}$ work, try all possible chaining variables that result from $h(k \oplus opad)$. Consider the common *kdf* construction described above with $HMAC()$ used for the hashing operation, $kdf(S,l) = [HMAC(S,P_1)||HMAC(S,P_2)|| \cdots ||HMAC(S,P_n)]_l$. The $P_1 \ldots P_n$ are known, and an attacker is given (or somehow acquires) a subset of the key, k_{P_i} corresponding to $HMAC(S,P_i)$, for some i. The attacker can try all possible chaining variables that could result from $h(k \oplus ipad)$, continue the hashing operation from each chaining variable (using $h()$) over the known data P_i, and obtain a value X. For each such X, the attacker can try all possible chaining variables that could result from $h(k \oplus opad)$, continue the hashing operation from each of these chaining variables (using $h()$) over X, and compare the result with k_{P_i}. Once the attacker finds a match, she knows that the correct inner and outer chaining variables have been found. All other bits of the key can then be determined trivially using these variables, the hash function $h()$, and the known data P_i.

The work factor to break this form of HMAC-based *kdf* is therefore $2^{2*hashlen}$, where *hashlen* is the output length of the HMAC hash function $h()$.

Thus, this common construction of a key derivation function from HMAC does not distribute the input entropy evenly over the output string either. Again, since the work factor here ($2^{2*hashlen}$) provides a security level which is likely sufficient for most applications, this is not an immediate (or even near-term) vulnerability. However, the lack of our desired property is again concerning and thus we recommend that HMAC should not be used in the construction of a key derivation function. We note that a similar attack is possible if $HMAC(P_i,S)$ is used instead of $HMAC(S,P_i)$.

3 Constructions That Overcome These Limitations

We supply two improved key derivation function constructions. The first does not distribute the input entropy evenly over the output string; however, it appears to be more secure than previous constructions. The second appears to have our desired property, but as yet we are unable to construct a rigourous proof that this is actually the case.

3.1 A Construction That Seems to Work

Going against the advice contained in Section 2.2, we provide an ad-hoc construction for which we have some concerns, but which appears to be more secure than the other constructions given. This construction may be referred to as a *computationally-secure kdf*, or *cs-kdf*, which may be appropriate for environments in which a more dedicated construction is undesirable or unnecessary.

Using the same notation as in Section 2.2, we define $kdf(S,l) = [h(S,P_1,S)|| h(S,P_2,S)|| \cdots ||h(S,P_n,S)]_l$.

Assume that S is a multiple of the hash function block length, B, and that the parameter strings are greater than B bits and can be influenced by the attacker. Now, for each of the $2^{hashlen}$ possible chaining variables resulting from the partial hash computation on the initial S, the attacker can attempt to find a

collision in the parameters, as in Section 2.2. This would require now $2^{hashlen} * 2^{\frac{hashlen}{2}} = 2^{\frac{3*hashlen}{2}}$ work, with the result being that for each of the $2^{hashlen}$ possible chaining variables, if the attacker is given $h(S, P_i, S)$ for a particular i then she can determine $h(S, P_j, S)$ for some $j \neq i$.

The problem is that since the attacker does not know S, she does not know the correct value of the chaining variable H. Thus, she must be able to obtain $O(2^{hashlen})$ values of $h(S, P_i, S)$ and look for a match. If one is obtained, then she will likely be able to determine $h(S, P_j, S)$. Strictly speaking, this construction then does not achieve our desired property against an attacker that is able to influence the parameters. However, since the attack also requires a large number of known hash outputs (larger than is ever achievable in practice), this construction appears to provide higher security than those described in Section 2.2.

3.2 An Improved Construction

The construction given in this section may be referred to as an *entropy-secure kdf*, or *es-kdf*. As we saw in Section 2.1, we are considering hash functions that have an internal chaining variable, H, of length *hashlen*. In order to begin processing the first block, this chaining variable is usually initialized to a fixed value H^0. Let us consider, for this construction, the hash function h such that the initial value of the chaining variable is allowed to vary as a parameter to the function. Thus we define the hash function $h_X()$, where X is the value of the initial chaining variable. The usual computation of $h()$ is thus written as $h_{H^0}()$.

Again, with the notation as in Section 2.2, let us define $L_1 = h_{H^0}(S)$, and $L_i = h_{L_{i-1}}(S)$ for $i > 1$. Our key derivation function can then be described as $kdf(S, l) = [L_1||L_2|| \cdots ||L_n]_l$.

Notice that the output of the hash function depends solely upon the input string S and the number of times that the hash function has been applied, and that only the string S gets input into the hash function directly. There is no dependence upon formatting the input in any particular way, or upon any parameter blocks or counters in the input data. We believe that, based upon the results in Section 2.1, it is a mistake and a security vulnerability to introduce such specified inputs into the construction.

3.3 Security

Intuitively, the attacks described in Section 2.2 do not apply to this construction since each application of the hash function to the input string S is processed completely differently by virtue of the change in the initial chaining variable each time.

Slightly more formally, let S' be the input string S padded (using the padding rule specified by the hash function $h()$) to the next input block length for $h()$. Then, the L_i above may be equivalently written as $L_i = h(\overbrace{S'||S'||S'|| \cdots ||S'}^{i \text{ times}})$.

Thus, L_j, $j > i$, may be written as $L_j = h(\overbrace{S'||\cdots||S'}^{i \text{ times}}||\overbrace{S'||\cdots||S'}^{j\text{-}i \text{ times}}) = h_{L_i}(\overbrace{S'||\cdots||S'}^{j\text{-}i \text{ times}}) = h_{L_i}(X)$, where $X = \overbrace{S'||\cdots||S'}^{j\text{-}i \text{ times}}$

Assuming that the initial value of the chaining variable does not affect the designed security properties of the hash function, given L_i, there are three ways to find L_j.

We can search for a collision that provides us with L_j. We could attempt to find $\overline{X}$ such that $h_{L_i}(\overline{X}) = L_j$ with $\overline{X} \neq X$. However, by assumption our attacker does not know L_j and thus finding this collision seems to be impossible. With $2^{\frac{hashlen}{2}}$ work though, our attacker can find a Y and $\overline{Y}$ such that $h_{L_i}(Y) = h_{L_i}(\overline{Y})$. This does not appear to be of any advantage since it is only with vanishing probability that $Y = X$ for some valid X corresponding to an L_j.

We can search for a pre-image of L_i. Assuming that S is generated by a process with e bits of entropy, there are at least 2^e possibilities for S. If $e \leq hashlen$ then determining other bits of the derived key from L_i is possible. If $e > hashlen$ then we would expect that there would still be at least $2^{e-hashlen}$ possible values for S that would provide the given L_i. Thus, this line of attack does not appear to provide any significant advantage in determining other bits of output of our key derivation function and therefore it still appears to have our desired property. Notice that the security of the full key derivation function output relies upon the entropy of S, which explains the name (*es-kdf*) of this construction.

We can find L_j through some other means from our knowledge of L_i. For example, perhaps it is possible that modifying the initial value of the chaining variable modifies the output of the hash function in some pre-determined way that is independent of the hash input. In other words, $h_{L_i}(Y) = f(h_{H^0}(Y))$ for all Y and some known $f()$. Then it would be possible to easily compute $L_{2i}, L_{3i}, \ldots$ simply from the knowledge of L_i. Given the flexibility that the presence of such a function would provide in choosing prefixes for known hashes, it seems that it may possibly allow a method for finding collisions (in which case such a function could not exist for collision-resistant hash functions). However, we are unable to prove that this is the case. Similarly, we do not know, and are unable to prove, whether it is possible to find L_j through any other means, given L_i.

With respect to the opposite direction, given L_j, it appears to be impossible to determine L_i. It seems reasonable that the presence of an algorithm to do so would violate the one-wayness of the hash function, but we do not yet have a proof for this conjecture.

An explicit proof that the *es-kdf* distributes the input entropy evenly over the output entropy is certainly desired and for further study. However, this construction does appear more amenable to a proof based upon solely the one-way and collision resistant properties of hash functions than previous constructions. Additionally, since it does not depend on formatting, parameter blocks, or counters, it appears to be superior to all previously proposed key derivation functions.

4 Commonly Used Key Derivation Functions

We now apply the analysis of Sections 2 and 3 to commonly used key derivation functions defined in popular standards.

4.1 The Concatenation Key Derivation Function

In the national and international standard specifications ANSI X9.42 [2], ANSI X9.63 [3], IEEE P1363a [12], ISO/IEC 18033-2 [13], and NIST Key Establishment Schemes [15], a construction referred to as the *Concatenation Key Derivation Function* is defined. Omitting some of the details (which may be found, for example, in X9.42 [2]), the construction can be described as follows. Compute $h_i = h(ZZ||Counter||[OtherInfo])$, where h_i is the i^{th} computation of the hash function $h()$, ZZ is a bit string denoting a shared secret value, $Counter$ is the counter value i represented as an octet string, and $OtherInfo$ is an optional bit string representing some (known) information shared between the communicating entities (such as data to specify the intended use of the derived key). Then $KeyingData$ is defined to be the leftmost $keylen$ bits of $h_1||h_2||\ldots||h_d$, where $keylen$ is an integer representing the length in bits of the keying data to be generated, and $d = \lceil keylen/hashlen \rceil$.

This X9.42 *Concatenation* construction is clearly a specific implementation of the general form given in Section 2.2 above and so suffers from the non-ideal characteristics of that form. The improved construction presented in Section 3.2 may be a better choice for high-security environments and applications.

4.2 The ASN.1 Key Derivation Function

In the national and international standard specifications ANSI X9.42 [2], NIST Key Establishment Schemes [15], and IETF S/MIME [18], another key derivation function is defined, called the *ASN.1 KDF*. This KDF is almost identical to the previous Concatenation KDF except that the format of all input and output data is specified in ASN.1. Thus, $h_i = h(ZZ||OtherInfo)$, where $OtherInfo$ consists of an algorithm identifier, a counter and other arbitrary information all placed within a particular ASN.1 structure. Then, as above, $KeyingData$ is the leftmost $keylen$ bits of $h_1||h_2||\ldots||h_d$.

Again, this construction is simply a specific case of the general construction of Section 2.2, and the improved construction of Section 3.2 may be a better choice in some high-security applications.

4.3 The Key Derivation Function in PKCS-5

In the specification PKCS-5 [17], the key derivation function PBKDF2 is defined as follows. Let $T_i = F(P, S, c, i)$ for $i = 1, \ldots, d$, where P is a secret password, S is a salt value, and c is an iteration count. The function $F()$ is defined to be $F(P, S, c, i) = U_1 \oplus U_2 \oplus \ldots U_c$, where $U_1 = prf(P, S||i)$ and $U_i = prf(P, U_{i-1})$ for $i = 2, \ldots, c$. The derived key is then the first $keylen$ bits of $T_1||T_2||\ldots||T_d$.

The example pseudo-random function prf given in this specification is HMAC-SHA-1, although it is noted that a hash function $h()$ may also meet the requirements under certain assumptions. For the purposes of this paper, we note that this is just a specific version of the constructions in Sections 2.2 and 2.3. In particular, if the length of the secret password P is such that it lies on the block boundary of the underlying hash function (i.e., $h()$ or SHA-1), then the work factor to break the derived key, given an appropriate subset of the key, will be $2^{hashlen}$ if the prf is $h()$, or 2^{320} if the prf is HMAC-SHA-1. In such situations, it may be preferable to use the *kdf* described in Section 3.2, although this is not strictly necessary: as observed in the specification, with password-based cryptography the search space for a password is unlikely to be greater than 160 bits.

4.4 The Key Derivation Function in TLS

The Transport Layer Security (TLS) Version 1 protocol specified in IETF RFC 2246 [5] defines a key derivation function $prf()$ in two pieces. First, a function called P_hash is defined, where

$$P\_hash(secret, seed) = HMAC\_hash(secret, A(1)||seed) \; || \\ HMAC\_hash(secret, A(2)||seed) \; || \ldots$$

and $A()$ is defined as $A(0) = seed$ and $A(i) = HMAC\_hash(secret, A(i-1))$. Second, an initial secret S is split into two pieces, S_1 and S_2, and the pseudo-random function is defined as $prf(secret, label, seed) = P\_MD5(S_1, label||seed) \oplus P\_SHA\text{-}1(S_2, label||seed)$ where $label$ is a pre-defined ASCII string. Keying material is produced by letting S be the $master\_secret$, $seed$ be the server's random value concatenated with the client's random value, and $label$ be the string "key expansion".

The use of HMAC to generate a long key stream can require $2^{2*hashlen}$ work to break, as discussed in Section 2.3. However, in the TLS construction, the use of both MD5 and SHA-1 means that the total work required to mount this type of attack will be at least $2^{2*128} * 2^{2*160} = 2^{576}$. Depending upon the entropy of the initial secret, the attacker may do better to simply try to guess the secret pieces S_1 and S_2. From the definitions of the MD5 and SHA-1 hash functions, these pieces may be up to 512 bits each, but for this specification they will be much smaller than this bound since the master secret is defined to be 48 bytes (384 bits) in length. The key derivation function defined for TLS, therefore, seems sufficiently strong for its purpose.

4.5 The Key Derivation Function in SPKM

The Simple Public-Key GSS-API Mechanism (SPKM) is specified in the IETF Proposed Standard RFC 2025 [1]. In this specification a k-bit key is generated as $rightmost\_k\_bits(OWF(context\_key||x||n||s||context\_key))$ where: x is the ASCII character C if the key is for a confidentiality algorithm, or the ASCII

character I if the key is for a keyed integrity algorithm; n is the number of the algorithm in the appropriate agreed list of algorithms for the context; s is the *stage* of processing, which is always the ASCII character 0 unless the key length k is greater than the output size of the One-Way Function OWF, in which case the OWF is computed repeatedly with increasing ASCII values of *stage* (each OWF output being concatenated to the end of previous OWF outputs), until k bits have been generated; and OWF is any appropriate One-Way Function.

This key derivation function is an example of the general form given in Section 3.1 above. While this is deemed to be adequate for most requirements, high-security environments may instead wish to use the construction of Section 3.2.

4.6 The Key Derivation Function in IKE

The Internet Key Exchange (IKE) specification, IETF RFC 2409 [10], contains a key derivation function defined as follows. Let $N_e = prf(N, C)$, where N is a nonce, C is a cookie containing predefined information (from the ISAKMP header), and $prf(key, msg)$ is a suitable keyed pseudo-random function (such as HMAC). Then, to generate a $keylen$-bit key K_e when $keylen$ is greater than $prflen$ (the output size of the prf), set K_e to be the most significant $keylen$ bits of K, where $K = K_1||K_2||\ldots||K_d$, $K_1 = prf(N_e, 0)$, and $K_i = prf(N_e, K_{i-1})$ for $i = 2, \ldots, d$. From this construction, the entropy of the process that generates K_e is at most $prflen$, since knowledge of either N and C, or N_e, suffices for an attacker to generate all of K. Equivalently, given any K_i, $0 \leq i < d$, a maximum of 2^{prflen} guesses at the value of N_e will allow an attacker to determine all subsequent bits of K (and therefore all subsequent bits of K_e). This clearly does not meet the goal of distributing the input entropy evenly over the output string. For environments that need $keylen > prflen$, then, it is recommended that the construction described in Section 3.2 be used directly on the N and C values.

5 Conclusion

We have examined the construction of key derivation functions based on cryptographic hash functions. A number of applications today require key sizes that are longer than the output length of commonly-used hash functions. In many implementations and in many national and international standard specifications, the hash function is iterated, with varying input data, to generate sufficient keying material. We have examined such constructions and found security flaws under certain conditions that may be concerning. We have also proposed new constructions and provided intuitive justifications for their security.

Acknowledgements. The authors would like to thank Hugo Krawczyk and the anonymous referees for their helpful comments.

References

1. Adams, C., "The Simple Public-Key GSS-API Mechanism (SPKM)", RFC 2025, October 1996.
2. ANSI X9.42-2001: *Public Key Cryptography For The Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography*, Accredited Standards Committee X9, 2001.
3. ANSI X9.63-2002: *Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography*, Accredited Standards Committee X9, 2002.
4. Bellare, M., R. Canetti, and H. Krawczyk, "Pseudorandom Functions Revisited: The Cascade Construction and its Concrete Security", `http://www-cse.ucsd.edu/users/mihir/papers/cascade.pdf` (see also *Proceedings of the 37th Symposium on Foundations of Computer Science*, IEEE, 1996, for an abridged version).
5. Dierks, T., and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.
6. FIPS 46-3, "Data Encryption Standard (DES)", *Federal Information Processing Standards Publication 46-3*, 1999. Available from `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.
7. FIPS 180-1, "Secure Hash Standard (SHS)", *Federal Information Processing Standards Publication 180-1*, 2002. Available from `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf`.
8. FIPS 197, "Advanced Encryption Standard (AES)", *Federal Information Processing Standards Publication 197*, 2001. Available from `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.
9. Freier, A., Karlton, P., and P. Kocher, "The SSL Protocol Version 3.0", draft-freier-ssl-version3-02.txt, November 18, 1996 (work in progress). Available at `http://wp.netscape.com/eng/ssl3/draft302.txt`.
10. Harkins, D., and D. Carrel, "The Internet Key Exchange (IKE)", RFC 2409, November 1998.
11. Håstad, J., Impagliazzo, R., Levin, L., and Luby, M., "A pseudorandom generator from any one-way function", *SIAM Journal on Computing*, **28** (1999), pp. 1364-1396.
12. IEEE P1363A: *Standard Specifications for Public Key Cryptography: Additional Techniques*, Institute of Electrical and Electronics Engineers, July 16, 2003, Draft Version 12.
13. ISO/IEC 18033-2, *Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric Ciphers*, Committee Draft, June 10, 2003.
14. Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.
15. National Institute of Standards and Technology, Special Publication 800-56: Recommendation On Key Establishment Schemes, Draft 2.0, January 2003.
16. National Institute of Standards and Technology, Special Publication 800-57: Recommendation For Key Management – Part 1: General Guideline, Draft, January 2003.
17. PKCS #5 v2.0, *Password-Based Cryptography Standard*, March 25, 1999. Available from `ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-5v2/pkcs5v2-0.pdf`.
18. Rescorla, E., "Diffie-Hellman Key Agreement Method", RFC 2631, June 1999.
19. Rivest, R., "The MD5 message-digest algorithm", RFC 1321, April 1992.
20. Yuval, G., "How to swindle Rabin", *Cryptologia*, **3** (1979), pp. 187-190.