Schedule of Fees (Sterling)

Citizen	Entries	Total Fee / per person (VAT included)			Fee detail				
					Visa fee Charged by the Chinese Embassy/Consulate General		Application service fee charged by the centre (VAT Included)		
		Regular applications	Express applications	Postal Service	Regular applications	Express applications	Regular applications	Express applications	Postal service
UK	Single entry	66	93	84	30	45	36	48	54
	Double entries	81	108	99	45	60	36	48	54
	Multiple entries 6 Months	126	153	144	90	105	36	48	54
	Multiple entries 12 Months	216	243	234	180	195	36	48	54
CANADA	Single entry	91	118	109	55	70	36	48	54
	Double entries								
	Multiple entries 6 Months								
	Multiple entries 12 Months								
US	Single entry	126	153	N/A	90	105	36	48	N/A
	Double entries								
	Multiple entries 6 Months								
	Multiple entries 12 Months								
Third country	Single entry	56	83	74	20	35	36	48	54
	Double entries	66	93	84	30	45	36	48	54
	Multiple entries 6 Months	76	103	94	40	55	36	48	54
	Multiple entries 12 Months	96	123	114	60	75	36	48	54

Notes: * Visa fees of Romanian, Serbian and Polish passport holders are charged in accordance with relevant regulations in Bilateral Agreements.

(1) Visa Fees are tax exempted and collected on behalf of the Chinese Embassy or Consulate General.

(2) Application Service Fees are charged by the Centre.

(3) The total amount of the fees to be paid by an applicant= Visa fee + Application Service fee + VAT (application service fee × VAT rate).

(4) The fees for applicants of third Countries may be different from those stated in the schedule.

STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS

COLIN PERCIVAL

Abstract. We introduce the concepts of memory-hard algorithms and sequential memory-hard functions, and argue that in order for key derivation functions to be maximally secure against attacks using custom hardware, they should be constructed from sequential memory-hard functions. We present a family of key derivation functions which, under the random oracle model of cryptographic hash functions, are provably sequential memory-hard, and a variation which appears to be marginally stronger at the expense of lacking provable strength. Finally, we provide some estimates of the cost of performing brute force attacks on a variety of password strengths and key derivation functions.

1. Introduction

Password-based key derivation functions are used for two primary purposes: First, to hash passwords so that an attacker who gains access to a password file does not immediately possess the passwords contained therewithin; and second, to generate cryptographic keys to be used for encrypting and/or authenticating data. While these two uses appear to be cryptologically quite different — in the first case, an attacker has the hash of a password and wishes to obtain the password itself, while in the second case, the attacker has data which is encrypted or authenticated with the password hash and wishes to obtain said password hash — they turn out to be effectively equivalent: Since all modern key derivation functions are constructed from hashes against which no non-trivial pre-image attacks are known, attacking the key derivation function directly is infeasible; consequently, the best attack in either case is to iterate through likely passwords and apply the key derivation function to each in turn.

Unfortunately, this form of "brute force" attack is quite liable to succeed. Users often select passwords which have far less entropy than is typically required of cryptographic keys; a recent study found that even for web sites such as `paypal.com`, where — since accounts are often linked to credit cards and bank accounts — one would expect users to make an effort to use strong passwords, the average password has an estimated entropy of 42.02 bits, while only a very small fraction had more than 64 bits of entropy [15]. In order to increase the cost of such brute force attacks, an approach known as "key stretching" or "key strengthening"[1] can be used:

E-mail address: `cperciva@tarsnap.com`.

[1] The phrase "key strengthening" was introduced by Abadi et al. [8] to refer to the process of adding additional entropy to a password in the form of a random suffix and verifying a password by conducting a brute-force search of possible suffixes; but the phrase is now widely used to mean the same thing as "key stretching".

By using a key derivation function which requires 2^s cryptographic operations to compute, the cost of performing a brute-force attack against passwords with t bits of entropy is raised from 2^t to 2^{s+t} operations [19].

This approach has been used with an increasing degree of formalism over the years. The original UNIX CRYPT function — dating back to the late 1970s — iterated the DES cipher 25 times in order to increase the cost of an attack [22], while Kamp's MD5-based hash [18] iterated the MD5 block cipher 1000 times; more recently, Provos and Mazières' bcrypt [24] and RSA Laboratories' PBKDF1 and PBKDF2 [17] are explicitly defined to perform a user-defined number of iterations[2], with the number of iterations presumably being stored along with the password salt.

Providing that the number of iterations used is increased as computer systems get faster, this allows legitimate users to spend a constant amount of time on key derivation without losing ground to attackers' ever-increasing computing power — as long as attackers are limited to the same software implementations as legitimate users. However, as Bernstein famously pointed out in the context of integer factorization [10], while parallelized hardware implementations may not change the number of *operations* performed compared to software implementations, this does not prevent them from dramatically changing the asymptotic *cost*, since in many contexts — including the embarrassingly parallel task of performing a brute-force search for a passphrase — dollar-seconds are the most appropriate units for measuring the cost of a computation[3]. As semiconductor technology develops, circuits do not merely become faster; they also become smaller, allowing for a larger amount of parallelism at the same cost. Consequently, using existing key derivation algorithms, even if the iteration count is increased such that the time taken to verify a password remains constant, the cost of finding a password by using a brute force attack implemented in hardware drops each year.

This paper aims to reduce the advantage which attackers can gain by using custom-designed parallel circuits.

2. Memory-hard algorithms

A natural way to reduce the advantage provided by an attacker's ability to construct highly parallel circuits is to increase the size of a single key derivation circuit — if a circuit is twice as large, only half as many copies can be placed on a given area of silicon — while still operating within the resources available to software implementations, including a powerful CPU and large amounts of RAM. Indeed, in the first paper to formalize the concept of key stretching [19] it is pointed out that requiring "32-bit arithmetic and use of moderately large amounts of RAM[4]" can make hardware attacks more expensive. However, widely used key derivation

[2]It should be noted, however, that when used to verify login passwords, the "user-defined" value is typically stored in a system configuration file which the vast majority of users never modify.

[3]That is, the price of hardware times the amount of time for which it needs to be used; this is analogous to the common AT (area times time) cost measure used in the context of VLSI circuit design. The ability of parallel designs to achieve a lower cost for the same number of operations is essentially due to their ability to use a larger fraction of die area for computational circuits.

[4]The example given is 256 32-bit words, which hardly qualifies as "moderately large" at the present time, and is questionable even in the context of hardware of the time (1997) given that even low-end PCs rarely had less than 4 MB of RAM (that being the official minimum requirement to run the Windows 95 operating system).

Schedule of Fees (Sterling)

Citizen	Entries	Total Fee / per person (VAT included)			Fee detail				
					Visa fee Charged by the Chinese Embassy/Consulate General		Application service fee charged by the centre (VAT Included)		
		Regular applications	Express applications	Postal Service	Regular applications	Express applications	Regular applications	Express applications	Postal service
UK	Single entry	66	93	84	30	45	36	48	54
	Double entries	81	108	99	45	60	36	48	54
	Multiple entries 6 Months	126	153	144	90	105	36	48	54
	Multiple entries 12 Months	216	243	234	180	195	36	48	54
CANADA	Single entry	91	118	109	55	70	36	48	54
	Double entries								
	Multiple entries 6 Months								
	Multiple entries 12 Months								
US	Single entry	126	153	N/A	90	105	36	48	N/A
	Double entries								
	Multiple entries 6 Months								
	Multiple entries 12 Months								
Third country	Single entry	56	83	74	20	35	36	48	54
	Double entries	66	93	84	30	45	36	48	54
	Multiple entries 6 Months	76	103	94	40	55	36	48	54
	Multiple entries 12 Months	96	123	114	60	75	36	48	54

Notes: * Visa fees of Romanian, Serbian and Polish passport holders are charged in accordance with relevant regulations in Bilateral Agreements.

(1) Visa Fees are tax exempted and collected on behalf of the Chinese Embassy or Consulate General.

(2) Application Service Fees are charged by the Centre.

(3) The total amount of the fees to be paid by an applicant= Visa fee + Application Service fee + VAT (application service fee × VAT rate).

(4) The fees for applicants of third Countries may be different from those stated in the schedule.

4: $b \leftarrow w_2 \mid \mathrm{0x00000001}$
5: $G \leftarrow w_4$
6: **for** $i = 0$ to 15 **do**
7:     $B_i \leftarrow 0$
8: **end for**
9: **for** $i = 0$ to $L - 1$ **do**
10:     $X \leftarrow (aX + b) \bmod 2^{32}$
11:     $V_i \leftarrow X + P_{i \bmod r} \bmod 2^{32}$
12: **end for**
13: **for** $n = 0$ to $N - 1$ **do**
14:     **for** $i = 0$ to $K - 1$ **do**
15:         $j \leftarrow G \bmod L$
16:         $G \leftarrow V_j$
17:         $X \leftarrow (aX + b) \bmod 2^{32}$
18:         $V_j \leftarrow X$
19:         $B_{i \bmod 16} \leftarrow B_{i \bmod 16} + G \bmod 2^{32}$
20:     **end for**
21:     **if** $n < r$ **then**
22:         $B_1 \leftarrow B_1 + P_n \bmod 2^{32}$
23:     **end if**
24:     $(w_0, w_1, w_2, w_3, w_4) \leftarrow \text{SHA1\_Compress}\left((w_0, w_1, w_2, w_3, w_4), B_0 \ldots B_{15}\right)$
25:     $X \leftarrow X + w_0 \bmod 2^{32}$
26:     $a \leftarrow w_1$
27:     $b \leftarrow w_2$
28:     $G \leftarrow w_4$
29: **end for**

There is a significant bug in this algorithm as stated above[7]: When the linear congruential generator is reinitialized on lines 25–27, there is no guarantee that the multiplier a is odd (unlike when the LCG is first initialized at lines 2–4); consequently, 50% of the time the LCG will rapidly degrade to the fixed point $b(1-a)^{-1} \bmod 2^{32}$. However, we do not believe that this error causes any significant reduction in the security of HEKS: If a large proportion of the entries in V are the same, then for even values of a the sequence of values of G in the inner loop (lines 14–20) will reach a fixed point shortly after the sequence of values of X; but for odd values of a, the sequence of values of G will not easily reach a fixed point. Consequently, in the algorithm as stated the vector V is likely to reach an equilibrium point where it has many duplicate entries but still contains significant entropy.

Reinhold suggests that the parameter K should be chosen to be $\sqrt{L}$ or larger, that L should be "as large as the user or user community can comfortably provide on the smallest machine on which they plan to use the algorithm", and that N should be determined as necessary to make the computation take the desired duration. Since HEKS takes, on a sequential machine, $O(L)$ space and $O(L + NK)$ time, it is memory-hard for $N = O(L^{1+\epsilon} K^{-1})$, e.g., if $N, K = O(\sqrt{L})$; and if the parameters are chosen as suggested, HEKS only fails to be memory-hard if there is not sufficient memory to match the desired duration of computation.

---

[7] Reinhold's description of the algorithm matches his C source code, so presumably this is not merely a typographical error.

However, this alone is not sufficient to eliminate the asymptotic advantage of an attacker armed with parallel hardware. On a parallel random access machine with L processors and $O(L)$ space, the next $\Omega(L^{0.5-\epsilon})$ outputs of the Bays-Durham PRNG can be computed in $O(\log L)$ time by applying binary powering to the permutation $j \leftarrow V_j \bmod L$ to compute the initial non-repeating sequence of values j; and the kth output of a linear congruential PRNG can be computed on a single processor in $O(\log k)$ time. Consequently, a parallel random access machine with $O(L)$ CPUs and $O(L)$ space can compute the HEKS key derivation function in $O(NKL^{-0.5+\epsilon})$ time, resulting in a computation cost of $O(NKL^{0.5+\epsilon})$ dollar-seconds — a very significant advantage over the $O(L^2 + NKL)$ cost of a naïve sequential implementation.

4. Sequential memory-hard functions

Clearly the inadequacy of HEKS as a key derivation function is due to its ability to effectively use multiple processors: While HEKS makes good use of a resource (RAM) which software implementations have in large supply, it can also make good use of a resource (computational parallelism) which is far more available in a hardware attack. To provide a framework for functions immune to this sort of attack, we introduce the following definition:

Definition 2. *A* sequential memory-hard function *is a function which*

(a) *can be computed by a memory-hard algorithm on a Random Access Machine in $T(n)$ operations; and*
(b) *cannot be computed on a Parallel Random Access Machine with $S^*(n)$ processors and $S^*(n)$ space in expected time $T^*(n)$ where $S^*(n)T^*(n) = O(T(n)^{2-x})$ for any $x > 0$.*

Put another way, a sequential memory-hard function is one where not only the fastest *sequential* algorithm is memory-hard, but additionally where it is impossible for a *parallel* algorithm to asymptotically achieve a significantly lower cost. Since memory-hard algorithms asymptotically come close to using the most space possible given their running time, and memory is the computationally usable resource general-purpose computers have which is most expensive to reproduce in hardware[8], we believe that, for any given running time on a sequential general-purpose computer, functions which are sequential memory-hard come close to being the most expensive possible functions to compute in hardware.

Indeed, it is surprising to consider the effect of adjusting parameters so that a sequential memory-hard function takes twice as long to compute: As long as there is enough random-access memory to compute the function on a general-purpose system, doubling the time spent asymptotically results in the cost of computing the function in hardware increasing *four-fold*, since the time and required space are both doubled.

[8]On many general-purpose systems, the CPU and motherboard are more expensive than the RAM; but the vast majority of that cost is due to the requirements of general-purpose computation, rapid sequential computation, and support for peripheral devices. The area occupied by *computation* logic on modern CPUs is vanishingly small compared to caches, instruction decoding, out-of-order execution, and I/O.

5. ROMix

The definition of sequential memory-hard functions, while theoretically interesting in its consequences, would not be of any practical value if it turned out that no sequential memory-hard functions exist; to that end, we introduce the class of functions $\text{ROMix}_H : \{0,1\}^k \times \{0 \dots 2^{k/8}-1\} \to \{0,1\}^k$ computed as follows[9]:

Algorithm ROMix$_H(B, N)$

Parameters:

	H	A hash function.
	k	Length of output produced by H, in bits.
	Integerify	A bijective function from $\{0,1\}^k$ to $\{0, \dots 2^k - 1\}$.

Input:

	B	Input of length k bits.
	N	Integer work metric, $< 2^{k/8}$

Output:

	B'	Output of length k bits.

Steps:

$X \leftarrow B$
for $i = 0$ to $N-1$ **do**
    $V_i \leftarrow X$
    $X \leftarrow H(X)$
end for
for $i = 0$ to $N-1$ **do**
    $j \leftarrow \text{Integerify}(X) \bmod N$
    $X \leftarrow H(X \oplus V_j)$
end for
$B' \leftarrow X$

This algorithm can be thought of as computing a large number of "random" values, and then accessing them "randomly" in order to ensure that they are all stored in Random Access Memory. Before we can prove anything more formally, we need a simple lemma concerning the iterated application of random oracles.

Lemma 1. *Suppose that two algorithms, Algorithm A and Algorithm B exist such that*

(1) *Algorithm A takes as input the integers N, M, and k, a k-bit value B, and an oracle $H : \{0,1\}^k \to \{0,1\}^k$, and produces a kM-bit output value $A_{N,M,k}(B,H)$; and*
(2) *Algorithm B takes as input the integers N, M, k, and x, with $0 \le x < N$, and $A_{N,M,k}(B,H)$; and operates on a system which can simultaneously consult M copies of the oracle H in unit time and perform any other computations instantaneously, to compute the value $H^x(B)$.*

Then if the values $H^0(B) \dots H^{N-1}(B)$ are distinct and $N < 2^{k/8}$, Algorithm B operates in expected time at least $\frac{N}{4M+2} - \frac{1}{2}$ for random oracles H, k-bit values B, and integers $x \in \{0 \dots N-1\}$.

[9] We expect that for reasons of performance and simplicity, implementors will restrict N to being a power of 2, in which case the function Integerify can be replaced by reading the first (or last) machine-length word from a k-bit block.

Proof. For notational simplicity, we consider N, M, and k to be fixed and omit them from variable subscripts.

Consider the Process B*, taking input $A(B,H)$, and operating on a machine which can simultaneously consult NM copies of the oracle H in unit time and perform any other computations instantaneously, defined as follows[10]: Execute Algorithm B for each integer $x \in \{0 \ldots N-1\}$ in parallel (using up to M oracles for each value x); after Algorithm B has completed for a value x and has returned the value $H^x(B)$, input that value to an oracle in the following step; finally, if any oracles are unneeded (due to Algorithm B not using all M oracles at some point, due to Algorithm B having finished, or because two or more instances of Algorithm B are inputting the same value into oracles) then input arbitrary unique values to them[11].

Now define $R_i(B,H) \in \{0,1\}^k$ for $i \leq N/M - 1$ to be the set of values input by Process B* to the NM oracles at time i, define $\bar{R}_i(B,H) = R_0(B,H) \cup R_1(B,H) \ldots R_i(B,H)$, and define $\bar{H}(B) = \{H^0(B), \ldots H^{N-1}(B)\}$. Clearly if Algorithm B computes $H^x(B)$ in time t for some B, H, x, then $H^x(B) \in R_i(B,H) \subset \bar{R}_i(B,H)$ for all $i \geq t$. We will proceed by bounding the expected size of $\bar{R}_i(B,H) \cap \bar{H}(B)$ for any process taking a kM bit input and operating on NM oracles.

Let $\bar{R}_{i-1}(B,H)$ and the values of H evaluated thereupon be fixed, and consider the probability, over random B, H, that $H^x(B) \in \bar{R}_i(B,H)$. Trivially, if $H^{x-1}(B) \in \bar{R}_{i-1}(B,H)$, then $P(H^x(B) \in \bar{R}_i(B,H)) \leq 1$ (since the probability of anything is at most 1); but if $H^{x-1}(B) \notin \bar{R}_{i-1}(B,H)$ then the value of H evaluated at $H^{x-1}(B)$ is entirely random; so $P(H^x(B) \in \bar{R}_i(B,H)) = \left|\bar{R}_i(B,H)\right| \cdot 2^{-k} = NM(i+1)2^{-k} \leq N^2 2^{-k}$. Now suppose that out of the $2^{2^k+1-NMi}$ values of (B,H) such that H takes the specified values, $s \cdot 2^{2^k+1-NMi}$ of them (i.e., a proportion s) result share the same value $A(B,H)$. Then the values of $\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|$ for such B, H are at most equal to the $s \cdot 2^{2^k+1-NMi}$ largest values of $\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|$ for permissible H.

However, the Chernoff bound states that for X a sum of independent 0-1 variables, $\mu = E(X)$ and Y a constant greater than μ,

$$P(X > Y) < \exp(Y - \mu + Y(\log \mu - \log Y)),$$

and so for $Y > 1$ we have

$$\begin{aligned} P(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right| - \left|\bar{R}_{i-1}(B,H) \cap \bar{H}(B)\right| > Y) & \\ < \exp(Y - N^3 2^{-k} + Y(\log(N^3 2^{-k}) - \log(Y))) & \\ < \exp(Y + Y \log(N^3 2^{-k})) & \\ = \left(eN^3 2^k\right)^Y < \left(2^{k/2}\right)^Y & \end{aligned}$$

and thus we find that, for (B,H) in the set of $s \cdot 2^{2^k+1-NMi}$ values such that $A(B,H)$ and H evaluated on $\bar{R}_{i-1}(B,H)$ take the correct values,

$$E(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|) < \left|\bar{R}_i(B,H) \cap \bar{H}(B)\right| + \log(s^{-1})/\log(2^{k/2}) + 1,$$

[10]We refer to the *Process* B* instead of the *Algorithm* B* since it neither produces output nor terminates.

[11]Thus for any B, H, Process B* inputs disjoint sets of NM values to the oracles at successive time steps.

where the $+1$ arises as a trivial upper bound on the contribution from the exponentially decreasing probabilities of values $X - Y$ for $X > Y$.

Now we note that $E(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|)$, with expectation taken over *all* values (B, H) is merely the average of the 2^{NMi+Mk} values $E(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|)$ with the expectation taken over (B, H) consistent with a given $A(B, H)$ and values of H at $\bar{R}_{i-1}(B, H)$; and by convexity, the resulting bound is weakest when all of the values s are equal, i.e., $s = 2^{-Mk}$. Consequently, we obtain (with the expectation taken over all (B, H))

$$\begin{aligned} E(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|) &< \left|\bar{R}_i(B,H) \cap \bar{H}(B)\right| + 2M + 1 \\ &< (2M+1) \cdot (i+1). \end{aligned}$$

The result now follows easily: Writing t_x as the expected time taken by Algorithm B to compute $H^x(B)$ and noting that the time it takes to compute $H^x(B)$ is equal to the number of sets $\bar{R}_i(B, H)$ which do not contain $H^x(B)$, we have

$$\begin{aligned} \frac{1}{N}\sum_{x=0}^{N-1} t_x &= \frac{1}{N}\sum_{i=0}^{\infty} N - E(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|) \\ &\geq \frac{1}{N}\sum_{i=0}^{\frac{N}{2M+1}-1} N - E(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|) \\ &> \frac{1}{N}\sum_{i=0}^{\frac{N}{2M+1}-1} N - (2M+1)(i+1) \\ &= \frac{N}{4M+2} - \frac{1}{2} \end{aligned}$$

as required.

□

While this proof is rather dense, the idea behind it is quite simple: If the value $H^{x-1}(B)$ has not yet been computed, there is no way to compute $H^x(B)$; and with only kM bits of information stored in $A(B, H)$, any algorithm will be limited to computing $O(M)$ values $H^x(B)$ from $A(B, H)$ directly and then iterating H to compute the rest. We believe that the "correct" lower bound on the expected running time is in fact $\frac{N}{2M} - \frac{1}{2}$, but this appears difficult to prove.

In spite of being marginally weaker than desirable, this lemma is still sufficient to prove the following theorem:

Theorem 1. *Under the Random Oracle model, the class of functions ROMix$_H$ are sequential memory-hard.*

Proof. The algorithm stated earlier uses $O(N)$ storage and operates in $O(N)$ time on a Random Access Machine, so clearly the functions can be computed by a memory-hard algorithm in $T(N) = O(N)$ operations.

Now suppose that ROMix$_H$ can be computed in $S^*(N) = M(N)$ space. Since H is a random oracle, it is impossible to compute the function without computing each of the values V_j and X in steps 6–9 of the sequential algorithm in turn; but by Lemma 1, it takes at least $O(N/M(N))$ time to compute each V_j.

Consequently, it takes at least $T^*(N) = O(N^2/M(N))$ time to compute the function, and thus $S^*(N)T^*(N) = O(N^2)$ as required. □

6. SMix

While ROMix performs well on a theoretical Random Access Machine, it suffers somewhat on real-world machines. In the real world, random access memory isn't; instead, factors such as caches, prefetching, and virtual memory[12] make small "random" memory accesses far more expensive than accesses to consecutive memory locations.

Existing widely used hash functions produce outputs of up to 512 bits (64 bytes), closely matching the cache line sizes of modern CPUs (typically 32–128 bytes), and the computing time required to hash even a very small amount of data (typically 200–2000 clock cycles on modern CPUs, depending on the hash used) is sufficient that the memory latency cost (typically 100–500 clock cycles) does not dominate the running time of ROMix.

However, as semiconductor technology advances, it is likely that neither of these facts will remain true. Memory latencies, measured in comparison to CPU performance or memory bandwidth, have been steadily increasing for decades, and there is no reason to expect that this will cease — to the contrary, switching delays impose a lower bound of $\Omega(\log N)$ on the latency of accessing a word in an N-byte RAM, while the speed of light imposes a lower bound of $\Omega(\sqrt{N})$ for 2-dimensional circuits. Furthermore, since most applications exhibit significant locality of reference, it is reasonable to expect cache designers to continue to increase cache line sizes in an attempt to trade memory bandwidth for (avoided) memory latency.

In order to avoid having ROMix become latency-limited in the future, it is necessary to apply it to larger hash functions. While we have only proved that ROMix is sequential memory-hard under the Random Oracle model, by considering the structure of the proof we note that the full strength of this model does not appear to be necessary. The critical properties of the hash function required in order for ROMix to be sequential memory-hard appear to be the following[13]:

(1) The outputs of H are uniformly distributed over $\{0,1\}^k$.
(2) It is impossible to iterate H quickly, even given access to many copies of the oracle and precomputation producing a limited-space intermediate.
(3) It is impossible to compute $\text{Integerify}(H(x))$ significantly faster than computing $H(x)$.

Most notably, there is no requirement that the function H have the usual properties of collision and pre-image resistance which are required of cryptographic hashes.

There are also two more criteria required of the hash function in order for ROMix to maximize the cost of a brute-force attack given an upper bound on the amount of computing time taken to compute the function in software:

(4) The ratio of the hash length k to the number of operations required to compute the hash function should be as large as possible.
(5) The hash function should not have significantly more internal parallelism than is available to software implementations.

[12] Even if data is stored in RAM, the first access to a page typically incurs a significant cost as the relevant paging tables are consulted.

[13] The first requirement limits the number of values $H^x(B)$ which $A(B,H)$ can uniquely identify; the second requirement ensures that values $H^x(B)$ which are not stored cannot be computed quickly; and the third requirement ensures that each iteration of the loop in lines 6–9 must complete before the next iteration starts.

In light of these, we define the function $\text{BlockMix}_{H,r}$ computed as follows:

Algorithm BlockMix$_{H,r}(B)$

Parameters:

	H	A hash function.
	r	Block size parameter

Input:

	$B_0 \dots B_{2r-1}$	Input vector of $2r$ k-bit blocks

Output:

	$B'_0 \dots B'_{2r-1}$	Output vector of $2r$ k-bit blocks.

Steps:

1: $X \leftarrow B_{2r-1}$
2: **for** $i = 0$ to $2r - 1$ **do**
3: $\quad X \leftarrow H(X \oplus B_i)$
4: $\quad Y_i \leftarrow X$
5: **end for**
6: $B' \leftarrow (Y_0, Y_2, \dots Y_{2r-2}, Y_1, Y_3, \dots Y_{2r-1})$

This function clearly satisfies condition (1) if the underlying H is uniformly distributed; it satisfies condition (3) if $\text{Integerify}(B_0 \dots B_{2r-1})$ is defined as a function of B_{2r-1}; and it is clearly optimal according to criteria (4) and (5) compared to any functions constructed out of the same underlying H. We conjecture that BlockMix also satisfies criteria (2), on the basis that the "shuffling" which occurs at step 6 should thwart any attempt to rapidly iterate BlockMix using precomputed values which uniquely identify some but not all of the values B_i; but this does not appear to be easily proven[14].

Given that the performance of BlockMix according to criteria (4) and (5) is exactly the same as the performance of the underlying hash H, BlockMix is best used with a hash which is fast while not possessing excess internal parallelism; based on this, it appears that Bernstein's Salsa20/8 core [11] is the best-performing widely studied cryptographic function available[15]. While Bernstein recommends using the Salsa20 core by adding diagonal constants [13] and uses it in this manner in his Salsa20 cipher and Rumba20 compression functions, we do not believe that this is necessary when the Salsa20 core is being used in ROMix and BlockMix, since the related-input attacks against which they defend are not relevant in this context.

Putting this together, we have the following:

Definition 3. *The function* $SMix_r : \{0,1\}^{1024r} \times \{0 \dots 2^{64} - 1\} \rightarrow \{0,1\}^{1024r}$ *is* $SMix_r(B, N) = ROMix_{BlockMix_{Salsa20/8,r}}(B, N)$ *where* $Integerify(B_0 \dots B_{2r-1})$ *is defined as the result of interpreting* B_{2r-1} *as a little-endian integer.*

Theorem 2. *The function* $SMix_r(B, N)$ *can be computed in* $4Nr$ *applications of the Salsa20/8 core using* $1024Nr + \mathrm{O}(r)$ *bits of storage.*

Proof. The above algorithms operate in the required time and space. $\square$

[14] If the shuffling is omitted from BlockMix, it can be rapidly iterated given precomputed values B_0, since the computations would neatly "pipeline".

[15] Bernstein's Chacha [12] appears to have a very slight advantage over Salsa20, but is newer and less widely used, and consequently has been less studied.

7. SCRYPT

Given a sequential memory-hard "mixing" function MF and a pseudorandom function PRF it is simple to construct a strong key derivation function. We define the class of functions $\mathrm{MFcrypt}_{PRF,MF}(P, S, N, p, dkLen)$ as computed by the following algorithm:

Algorithm MFcrypt$_{H,MF}(P, S, N, p, dkLen)$

Parameters:

PRF	A pseudorandom function.
$hLen$	Length of output produced by PRF, in octets.
MF	A sequential memory-hard function from $\mathbb{Z}_{256}^{MFLen} \times \mathbb{N}$ to $\mathbb{Z}_{256}^{MFLen}$.
$MFLen$	Length of block mixed by MF, in octets.

Intput:

P	Passphrase, an octet string.
S	Salt, an octet string.
N	CPU/memory cost parameter.
p	Parallelization parameter; a positive integer satisfying $p \leq (2^{32} - 1)hLen/MFLen$.
$dkLen$	Intended output length in octets of the derived key; a positive integer satisfying $dkLen \leq (2^{32} - 1)hLen$.

Output:

DK	Derived key, of length $dkLen$ octets.

Steps:

1: $(B_0 \ldots B_{p-1}) \leftarrow \mathrm{PBKDF2}_{PRF}(P, S, 1, p \cdot MFLen)$
2: **for** $i = 0$ to $p - 1$ **do**
3: $\quad B_i \leftarrow MF(B_i, N)$
4: **end for**
5: $DK \leftarrow \mathrm{PBKDF2}_{PRF}(P, B_0 \parallel B_1 \parallel \ldots \parallel B_{p-1}, 1, dkLen)$

This algorithm uses PBKDF2 [17] with the pseudorandom function PRF to generate p blocks of length $MFLen$ octets from the provided password and salt; these are independently mixed using the mixing function MF; and the final output is then generated by applying PBKDF2 once again, using the well-mixed blocks as salt[16]. Since, for large N, the calls to MF take asymptotically longer than the calls to PBKDF2, and the blocks B_i produced using PBKDF2 are independent and random, subject to H being a random oracle, we note that if MF is a sequential memory-hard function then MFcrypt is sequential memory-hard under the random oracle model.

We now apply MFcrypt to the mixing function SMix from the previous section and the SHA256 hash function:

Definition 4. *The key derivation function scrypt is defined as*

$$scrypt(P, S, N, r, p, dkLen) = MFcrypt_{HMAC\_SHA256, SMix_r}(P, S, N, p, dkLen)$$

[16]The limits on the size of p and $dkLen$ exist as a result of a corresponding limit on the length of key produced by PBKDF2.

Users of scrypt can tune the parameters N, r, and p according to the amount of memory and computing power available, the latency-bandwidth product of the memory subsystem, and the amount of parallelism desired; at the current time, taking $r = 8$ and $p = 1$ appears to yield good results, but as memory latency and CPU parallelism increase it is likely that the optimum values for both r and p will increase. Note also that since the computations of SMix are independent, a large value of p can be used to increase the computational cost of scrypt without increasing the memory usage; so we can expect scrypt to remain useful even if the growth rates of CPU power and memory capacity diverge.

Conjecture 1. *If it is impossible for a circuit to compute the Salsa20/8 core in less than t time, and it is impossible for a circuit to store x bits of data in less than sx area for any $x \geq 0$, then it is impossible to compute scrypt$(P, S, N, r, p, dkLen)$ in a circuit with an expected amortized area-time product per password of less than $1024N^2r^2pst$.*

Put simply, this conjecture states that combining MFcrypt, ROMix, BlockMix, and the Salsa20/8 core does not expose scrypt to any attacks more powerful than the "generic" algorithms for computing ROMix.

8. Brute-force attack costs

Given a set of key derivation functions, it is natural to ask how much it would cost an attacker to perform a brute-force search over a class of passwords in order to find a particular password given its hash (or, equivalently, given some cryptographic ciphertext which can be used to quickly accept or reject potential password hashes). It is difficult to obtain accurate data concerning the cost of hardware password-cracking circuits — those few organizations which have the resources and inclination to design and fabricate custom circuits for password-cracking tend to be somewhat secretive — and so we must rely instead on estimating the costs of the underlying cryptographic operations in the expectation that the other costs are comparatively negligible. Even given this approximation the amount of information available is limited, since much of the work of implementing cryptographic circuits has been performed by private corporations which have clear financial reasons to restrict access to information about their products to potential customers.

Based on available data concerning DES [1, 4, 5], MD5 [2, 6], Blowfish [14, 21], SHA-256 [3, 7, 20], and Salsa20 [16, 27] cores, we provide the following estimates for the size and performance of cryptographic circuits on a 130 nm process[17]:

- A DES circuit with ≈ 4000 gates of logic can encrypt data at 2000 Mbps.
- An MD5 circuit with ≈ 12000 gates of logic can hash data at 2500 Mbps.
- A SHA256 circuit with ≈ 20000 gates of logic can hash data at 2500 Mbps.
- A Blowfish circuit with ≈ 22000 gates of logic and 4 kiB of SRAM can encrypt data at 1000 Mbps.
- A Salsa20/8 circuit with ≈ 24000 gates of logic can output a keystream at 2000 Mbps.

We also make estimates of the cost of manufacturing integrated circuits on a 130 nm process circa 2002:

[17] We use 130 nm as a basis for comparison simply because this is the process technology for which the most information was readily available concerning cryptographic circuits.

- Each gate of random logic requires $\approx 5\ \mu\text{m}^2$ of VLSI area.
- Each bit of SRAM requires $\approx 2.5\ \mu\text{m}^2$ of VLSI area.
- Each bit of DRAM requires $\approx 0.1\ \mu\text{m}^2$ of VLSI area.
- VLSI circuits cost $\approx 0.1\$/\text{mm}^2$.

Using these values, we estimate the cost of computing 9 key derivation functions: the original CRYPT; the MD5 hash (which, although not designed for use as a key derivation function, is nonetheless used as such by many applications); Kamp's MD5-based hash; PBKDF2-HMAC-SHA256 with an iteration count of 86,000; PBKDF2-HMAC-SHA256 with an iteration count of 4,300,000; bcrypt with $cost = 11$; bcrypt with $cost = 16$; scrypt with $(N, r, p) = (2^{14}, 8, 1)$; and scrypt with $(N, r, p) = (2^{20}, 8, 1)$. For the parameterized KDFs the parameters are chosen such that the running time on one core of a 2.5 GHz Intel Core 2 Duo processor[18] is less than 100 ms (for the lower parameters) or less than 5 s (for the higher parameters); we chose these values since 100 ms is a reasonable upper bound on the delay which should be cryptographically imposed on interactive logins, while 5 s is a reasonable amount of time to be spent encrypting or decrypting a sensitive file.

For each key derivation function, we consider six different types of password:

- A random sequence of 6 lower-case letters; e.g., "`sfgroy`".
- A random sequence of 8 lower-case letters; e.g., "`ksuvnwyf`".
- A random sequence of 8 characters selected from the 95 printable 7-bit ASCII characters; e.g., "`6,uh3y[a`".
- A random sequence of 10 characters selected from the 95 printable 7-bit ASCII characters; e.g., "`H.*W8Jz&r3`".
- A 40-character string of text; e.g., "`This is a 40-character string of English`".
- An 80-character string of text; e.g., "`This is an 80-character phrase which you probably won't be able to crack easily.`".

For the strings of text, we estimate entropy following the guidance provided by NIST [23]: The first character is taken to have 4 bits of entropy, the next 7 characters are taken to have 2 bits of entropy each, the following 12 characters are taken to have 1.5 bits of entropy each, and subsequent characters are taken to have 1 bit of entropy each.

In Table 1 we show the estimated costs of "cracking" hashed passwords in dollar-years; or equivalently, the cost of hardware which can find a password in an average time of 1 year (i.e., which would take 2 years to search the complete password space). We caution again that these values are very approximate and reflect only the cost of the cryptographic circuitry with circa 2002 technology: It is quite possible that the costs of other hardware (control circuitry, boards, power supplies) and operating costs (power, cooling) would increase the costs by a factor of 10 above these; and it is equally possible that improvements in semiconductor technology and improved cryptographic circuit designs could each reduce the costs by a factor of 10. Nevertheless, we believe that the estimates presented here are useful for the purpose of comparing different key derivation functions.

It is clear from this table that scrypt is a much more expensive key derivation function to attack than the alternatives: When used for interactive logins, it is 35 times more expensive than bcrypt and 260 times more expensive than PBKDF2;

[18] This processor is also known as "the CPU in the author's laptop".

TABLE 1. Estimated cost of hardware to crack a password in 1 year.

KDF	6 letters	8 letters	8 chars	10 chars	40-char text	80-char text
DES CRYPT	< \$1	< \$1	< \$1	< \$1	< \$1	< \$1
MD5	< \$1	< \$1	< \$1	\$1.1k	\$1	\$1.5T
MD5 CRYPT	< \$1	< \$1	\$130	\$1.1M	\$1.4k	$\$1.5 \times 10^{15}$
PBKDF2 (100 ms)	< \$1	< \$1	\$18k	\$160M	\$200k	$\$2.2 \times 10^{17}$
bcrypt (95 ms)	< \$1	\$4	\$130k	\$1.2B	\$1.5M	\$48B
scrypt (64 ms)	< \$1	\$150	\$4.8M	\$43B	\$52M	$\$6 \times 10^{19}$
PBKDF2 (5.0 s)	< \$1	\$29	\$920k	\$8.3B	\$10M	$\$11 \times 10^{18}$
bcrypt (3.0 s)	< \$1	\$130	\$4.3M	\$39B	\$47M	\$1.5T
scrypt (3.8 s)	\$900	\$610k	\$19B	\$175T	\$210B	$\$2.3 \times 10^{23}$

and when used for file encryption — where, unlike bcrypt and PBKDF2, scrypt uses not only more CPU time but also increases the die area required — scrypt increases its lead to a factor of 4000 over bcrypt and 20000 over PBKDF2. It is also worth noting that while bcrypt is stronger than PBKDF2 for most types of passwords, it falls behind for long passphrases; this results from bcrypt's inability to use more than the first 55 characters of a passphrase[19]. While our estimated costs and NIST's estimates of passphrase entropy suggest that bcrypt's 55-character limitation is not likely to cause problems at the present time, implementors of systems which rely on bcrypt might be well-advised to either work around this limitation (e.g., by "pre-hashing" a passphrase to make it fit into the 55-character limit) or to take steps to prevent users from placing too much password entropy in the 56th and subsequent characters (e.g., by asking users of a website to type their password into an input box which only has space for 55 characters).

9. Conclusions

We have proven that, under the random oracle model, the mixing function ROMix_H is sequential memory-hard; and it appears very likely that the scrypt key derivation function is also sequential memory-hard. Providing that no new attacks on scrypt or its underlying components are found, a brute-force attack on scrypt is many times harder than similar attacks on other key derivation functions; consequently, we recommend that implementors of new cryptographic systems should strongly consider using scrypt.

Finally, we recommend that cryptographic consumers make themselves aware of the strengths of the key derivation functions they are using, and choose passwords accordingly; we suspect that even generally security-conscious users are in many cases not aware how (in)secure their passwords are.

10. Acknowledgements

We thank Arnold G. Reinhold, Daniel J. Bernstein, Graeme Durant, and Paul Kocher for the advice and information they have provided.

[19]This is, however, far better than the original DES-based CRYPT, which only hashed the first 8 bytes of a password and is consequently absurdly cheap to break, regardless of the underlying password distribution.

References

[1] CAST DES data encryption standard core. http://www.cast-inc.com/cores/des/cast_des.pdf.

[2] CAST MD5 hash function core. http://www.cast-inc.com/cores/md5/cast_md5.pdf.

[3] CAST SHA256 secure hash function core. http://www.cast-inc.com/cores/sha-256/cast_sha256.pdf.

[4] DES1 ultra-compact data encryption standard (DES/3DES) core. http://www.ipcores.com/DES1core.htm.

[5] Helion technology datasheet - high performance DES and triple-DES core for asic. http://www.heliontech.com/downloads/des_asic_helioncore.pdf.

[6] Helion technology datasheet - high performance MD5 hash core for asic. http://www.heliontech.com/downloads/md5_asic_helioncore.pdf.

[7] Helion technology datasheet - high performance SHA-256 hash core for asic. http://www.heliontech.com/downloads/sha256_asic_fast_helioncore.pdf.

[8] M. Abadi, T. Mark, A. Lomas, and R. Needham. Strengthening passwords. Technical report, SRC Technical Note, 1997.

[9] C. Bays and S.D. Durham. Improving a poor random number generator. *ACM transactions on mathematical software*, 2(1):59–64, 1976.

[10] D.J. Bernstein. Circuits for integer factorization: a proposal, 2001. http://cr.yp.to/papers.html#nfscircuit.

[11] D.J. Bernstein. The Salsa20 family of stream ciphers, 2007. http://cr.yp.to/papers.html#salsafamily.

[12] D.J. Bernstein. ChaCha, a variant of Salsa20, 2008. http://cr.yp.to/papers.html#chacha.

[13] D.J. Bernstein. Personal communication, 2009.

[14] G. Durant. Personal communication, 2009.

[15] D. Florêncio and C. Herley. A large-scale study of web password habits. In *WWW '07: Proc. of the 16th international World Wide Web conference*, pages 657–666, 2007.

[16] T. Good and M. Benaissa. Hardware results for selected stream cipher candidates. In *Proc. of The State of the Art of Stream Ciphers*, 2007.

[17] B. Kaliski. PKCS #5: Password-based cryptography specification version 2.0. RFC 2898, 2000.

[18] P.-H. Kamp. MD5 crypt. FreeBSD 2.0, 1994. http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/lib/libcrypt/crypt.c?rev=1.2.

[19] J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Secure applications of low-entropy keys. In *ISW '97: Proc. of the first international workshop on information security*, pages 121–134, 1998.

[20] Y.K. Lee, H. Chan, and I. Verbauwhede. Iteration bound analysis and throughput optimum architecture of SHA-256 (384, 512) for hardware implementations. In *Workshop on Information Security Applications 2007*, LNCS 4867, 2008.

[21] M.C.-J. Lin and Y.-L. Lin. A VLSI implementation of the blowfish encryption/decryption algorithm. In *ASP-DAC '00: Proceedings of the 2000 conference on Asia South Pacific design automation*, 2000.

[22] R.H. Morris and K. Thompson. UNIX password security. *Communications of the ACM*, 22(11), 1979.

[23] National Institute of Standards and Technology. Electronic authentication guideline. NIST Special Publication 800-63, 2006.

[24] N. Provos and D. Mazières. A future-adaptable password scheme. In *Proc. of the FREENIX track: 1999 USENIX annual technical conference*, 1999.

[25] A.G. Reinhold. HEKS: A family of key stretching algorithms, 1999. http://world.std.com/~reinhold/HEKSproposal.html.

[26] A.G. Reinhold. Personal communication, 2009.

[27] J. Yan and H.M. Heys. Hardware implementation of the Salsa20 and Phelix stream ciphers. In *Proc. of the IEEE Canadian Conference on Electrical and Computer Engineering*, 2007.

Appendix A. Availability

Source code for scrypt, including reference and optimized implementations in C, and a demonstration file-encryption utility are available for download and use under the 2-clause BSD license from `http://www.tarsnap.com/scrypt/`.

Appendix B. Test vectors

For reference purposes, we provide the following test vectors for scrypt, where the password and salt strings are passed as sequences of ASCII bytes without a terminating NUL:

scrypt("", "", 16, 1, 1, 64) =

```
77 d6 57 62 38 65 7b 20 3b 19 ca 42 c1 8a 04 97
f1 6b 48 44 e3 07 4a e8 df df fa 3f ed e2 14 42
fc d0 06 9d ed 09 48 f8 32 6a 75 3a 0f c8 1f 17
e8 d3 e0 fb 2e 0d 36 28 cf 35 e2 0c 38 d1 89 06
```

scrypt("password", "NaCl", 1024, 8, 16, 64) =

```
fd ba be 1c 9d 34 72 00 78 56 e7 19 0d 01 e9 fe
7c 6a d7 cb c8 23 78 30 e7 73 76 63 4b 37 31 62
2e af 30 d9 2e 22 a3 88 6f f1 09 27 9d 98 30 da
c7 27 af b9 4a 83 ee 6d 83 60 cb df a2 cc 06 40
```

scrypt("pleaseletmein", "SodiumChloride", 16384, 8, 1, 64) =

```
70 23 bd cb 3a fd 73 48 46 1c 06 cd 81 fd 38 eb
fd a8 fb ba 90 4f 8e 3e a9 b5 43 f6 54 5d a1 f2
d5 43 29 55 61 3f 0f cf 62 d4 97 05 24 2a 9a f9
e6 1e 85 dc 0d 65 1e 40 df cf 01 7b 45 57 58 87
```

scrypt("pleaseletmein", "SodiumChloride", 1048576, 8, 1, 64) =

```
21 01 cb 9b 6a 51 1a ae ad db be 09 cf 70 f8 81
ec 56 8d 57 4a 2f fd 4d ab e5 ee 98 20 ad aa 47
8e 56 fd 8f 4b a5 d0 9f fa 1c 6d 92 7c 40 f4 c3
37 30 40 49 e8 a9 52 fb cb f4 5c 6f a7 7a 41 a4
```

STRONGER KEY DERIVATION VIA SEQUENTIAL MEMORY-HARD FUNCTIONS

COLIN PERCIVAL

Abstract. We introduce the concepts of memory-hard algorithms and sequential memory-hard functions, and argue that in order for key derivation functions to be maximally secure against attacks using custom hardware, they should be constructed from sequential memory-hard functions. We present a family of key derivation functions which, under the random oracle model of cryptographic hash functions, are provably sequential memory-hard, and a variation which appears to be marginally stronger at the expense of lacking provable strength. Finally, we provide some estimates of the cost of performing brute force attacks on a variety of password strengths and key derivation functions.

1. Introduction

Password-based key derivation functions are used for two primary purposes: First, to hash passwords so that an attacker who gains access to a password file does not immediately possess the passwords contained therewithin; and second, to generate cryptographic keys to be used for encrypting and/or authenticating data. While these two uses appear to be cryptologically quite different — in the first case, an attacker has the hash of a password and wishes to obtain the password itself, while in the second case, the attacker has data which is encrypted or authenticated with the password hash and wishes to obtain said password hash — they turn out to be effectively equivalent: Since all modern key derivation functions are constructed from hashes against which no non-trivial pre-image attacks are known, attacking the key derivation function directly is infeasible; consequently, the best attack in either case is to iterate through likely passwords and apply the key derivation function to each in turn.

Unfortunately, this form of "brute force" attack is quite liable to succeed. Users often select passwords which have far less entropy than is typically required of cryptographic keys; a recent study found that even for web sites such as `paypal.com`, where — since accounts are often linked to credit cards and bank accounts — one would expect users to make an effort to use strong passwords, the average password has an estimated entropy of 42.02 bits, while only a very small fraction had more than 64 bits of entropy [15]. In order to increase the cost of such brute force attacks, an approach known as "key stretching" or "key strengthening"[1] can be used:

E-mail address: `cperciva@tarsnap.com`.

[1]The phrase "key strengthening" was introduced by Abadi et al. [8] to refer to the process of adding additional entropy to a password in the form of a random suffix and verifying a password by conducting a brute-force search of possible suffixes; but the phrase is now widely used to mean the same thing as "key stretching".

By using a key derivation function which requires 2^s cryptographic operations to compute, the cost of performing a brute-force attack against passwords with t bits of entropy is raised from 2^t to 2^{s+t} operations [19].

This approach has been used with an increasing degree of formalism over the years. The original UNIX CRYPT function — dating back to the late 1970s — iterated the DES cipher 25 times in order to increase the cost of an attack [22], while Kamp's MD5-based hash [18] iterated the MD5 block cipher 1000 times; more recently, Provos and Mazières' bcrypt [24] and RSA Laboratories' PBKDF1 and PBKDF2 [17] are explicitly defined to perform a user-defined number of iterations[2], with the number of iterations presumably being stored along with the password salt.

Providing that the number of iterations used is increased as computer systems get faster, this allows legitimate users to spend a constant amount of time on key derivation without losing ground to attackers' ever-increasing computing power — as long as attackers are limited to the same software implementations as legitimate users. However, as Bernstein famously pointed out in the context of integer factorization [10], while parallelized hardware implementations may not change the number of *operations* performed compared to software implementations, this does not prevent them from dramatically changing the asymptotic *cost*, since in many contexts — including the embarrassingly parallel task of performing a brute-force search for a passphrase — dollar-seconds are the most appropriate units for measuring the cost of a computation[3]. As semiconductor technology develops, circuits do not merely become faster; they also become smaller, allowing for a larger amount of parallelism at the same cost. Consequently, using existing key derivation algorithms, even if the iteration count is increased such that the time taken to verify a password remains constant, the cost of finding a password by using a brute force attack implemented in hardware drops each year.

This paper aims to reduce the advantage which attackers can gain by using custom-designed parallel circuits.

2. Memory-hard algorithms

A natural way to reduce the advantage provided by an attacker's ability to construct highly parallel circuits is to increase the size of a single key derivation circuit — if a circuit is twice as large, only half as many copies can be placed on a given area of silicon — while still operating within the resources available to software implementations, including a powerful CPU and large amounts of RAM. Indeed, in the first paper to formalize the concept of key stretching [19] it is pointed out that requiring "32-bit arithmetic and use of moderately large amounts of RAM[4]" can make hardware attacks more expensive. However, widely used key derivation

[2]It should be noted, however, that when used to verify login passwords, the "user-defined" value is typically stored in a system configuration file which the vast majority of users never modify.

[3]That is, the price of hardware times the amount of time for which it needs to be used; this is analogous to the common AT (area times time) cost measure used in the context of VLSI circuit design. The ability of parallel designs to achieve a lower cost for the same number of operations is essentially due to their ability to use a larger fraction of die area for computational circuits.

[4]The example given is 256 32-bit words, which hardly qualifies as "moderately large" at the present time, and is questionable even in the context of hardware of the time (1997) given that even low-end PCs rarely had less than 4 MB of RAM (that being the official minimum requirement to run the Windows 95 operating system).

Schedule of Fees (Sterling)

Citizen	Entries	Total Fee / per person (VAT included)			Fee detail				
					Visa fee Charged by the Chinese Embassy/Consulate General		Application service fee charged by the centre (VAT Included)		
		Regular applications	Express applications	Postal Service	Regular applications	Express applications	Regular applications	Express applications	Postal service
UK	Single entry	66	93	84	30	45	36	48	54
	Double entries	81	108	99	45	60	36	48	54
	Multiple entries 6 Months	126	153	144	90	105	36	48	54
	Multiple entries 12 Months	216	243	234	180	195	36	48	54
CANADA	Single entry Double entries Multiple entries 6 Months Multiple entries 12 Months	91	118	109	55	70	36	48	54
US	Single entry Double entries Multiple entries 6 Months Multiple entries 12 Months	126	153	N/A	90	105	36	48	N/A
Third country	Single entry	56	83	74	20	35	36	48	54
	Double entries	66	93	84	30	45	36	48	54
	Multiple entries 6 Months	76	103	94	40	55	36	48	54
	Multiple entries 12 Months	96	123	114	60	75	36	48	54

Notes: * Visa fees of Romanian, Serbian and Polish passport holders are charged in accordance with relevant regulations in Bilateral Agreements.

(1) Visa Fees are tax exempted and collected on behalf of the Chinese Embassy or Consulate General.

(2) Application Service Fees are charged by the Centre.

(3) The total amount of the fees to be paid by an applicant= Visa fee + Application Service fee + VAT (application service fee × VAT rate).

(4) The fees for applicants of third Countries may be different from those stated in the schedule.

4: $b \leftarrow w_2 \mid \text{0x00000001}$
5: $G \leftarrow w_4$
6: **for** $i = 0$ to 15 **do**
7:     $B_i \leftarrow 0$
8: **end for**
9: **for** $i = 0$ to $L - 1$ **do**
10:     $X \leftarrow (aX + b) \bmod 2^{32}$
11:     $V_i \leftarrow X + P_{i \bmod r} \bmod 2^{32}$
12: **end for**
13: **for** $n = 0$ to $N - 1$ **do**
14:     **for** $i = 0$ to $K - 1$ **do**
15:         $j \leftarrow G \bmod L$
16:         $G \leftarrow V_j$
17:         $X \leftarrow (aX + b) \bmod 2^{32}$
18:         $V_j \leftarrow X$
19:         $B_{i \bmod 16} \leftarrow B_{i \bmod 16} + G \bmod 2^{32}$
20:     **end for**
21:     **if** $n < r$ **then**
22:         $B_1 \leftarrow B_1 + P_n \bmod 2^{32}$
23:     **end if**
24:     $(w_0, w_1, w_2, w_3, w_4) \leftarrow \text{SHA1\_Compress}\left((w_0, w_1, w_2, w_3, w_4), B_0 \ldots B_{15}\right)$
25:     $X \leftarrow X + w_0 \bmod 2^{32}$
26:     $a \leftarrow w_1$
27:     $b \leftarrow w_2$
28:     $G \leftarrow w_4$
29: **end for**

There is a significant bug in this algorithm as stated above[7]: When the linear congruential generator is reinitialized on lines 25–27, there is no guarantee that the multiplier a is odd (unlike when the LCG is first initialized at lines 2–4); consequently, 50% of the time the LCG will rapidly degrade to the fixed point $b(1-a)^{-1} \bmod 2^{32}$. However, we do not believe that this error causes any significant reduction in the security of HEKS: If a large proportion of the entries in V are the same, then for even values of a the sequence of values of G in the inner loop (lines 14–20) will reach a fixed point shortly after the sequence of values of X; but for odd values of a, the sequence of values of G will not easily reach a fixed point. Consequently, in the algorithm as stated the vector V is likely to reach an equilibrium point where it has many duplicate entries but still contains significant entropy.

Reinhold suggests that the parameter K should be chosen to be $\sqrt{L}$ or larger, that L should be "as large as the user or user community can comfortably provide on the smallest machine on which they plan to use the algorithm", and that N should be determined as necessary to make the computation take the desired duration. Since HEKS takes, on a sequential machine, $O(L)$ space and $O(L + NK)$ time, it is memory-hard for $N = O(L^{1+\epsilon}K^{-1})$, e.g., if $N, K = O(\sqrt{L})$; and if the parameters are chosen as suggested, HEKS only fails to be memory-hard if there is not sufficient memory to match the desired duration of computation.

[7] Reinhold's description of the algorithm matches his C source code, so presumably this is not merely a typographical error.

However, this alone is not sufficient to eliminate the asymptotic advantage of an attacker armed with parallel hardware. On a parallel random access machine with L processors and $O(L)$ space, the next $\Omega(L^{0.5-\epsilon})$ outputs of the Bays-Durham PRNG can be computed in $O(\log L)$ time by applying binary powering to the permutation $j \leftarrow V_j \bmod L$ to compute the initial non-repeating sequence of values j; and the kth output of a linear congruential PRNG can be computed on a single processor in $O(\log k)$ time. Consequently, a parallel random access machine with $O(L)$ CPUs and $O(L)$ space can compute the HEKS key derivation function in $O(NKL^{-0.5+\epsilon})$ time, resulting in a computation cost of $O(NKL^{0.5+\epsilon})$ dollar-seconds — a very significant advantage over the $O(L^2 + NKL)$ cost of a naïve sequential implementation.

4. Sequential memory-hard functions

Clearly the inadequacy of HEKS as a key derivation function is due to its ability to effectively use multiple processors: While HEKS makes good use of a resource (RAM) which software implementations have in large supply, it can also make good use of a resource (computational parallelism) which is far more available in a hardware attack. To provide a framework for functions immune to this sort of attack, we introduce the following definition:

Definition 2. *A* sequential memory-hard function *is a function which*

(a) *can be computed by a memory-hard algorithm on a Random Access Machine in $T(n)$ operations; and*
(b) *cannot be computed on a Parallel Random Access Machine with $S^*(n)$ processors and $S^*(n)$ space in expected time $T^*(n)$ where $S^*(n)T^*(n) = O(T(n)^{2-x})$ for any $x > 0$.*

Put another way, a sequential memory-hard function is one where not only the fastest *sequential* algorithm is memory-hard, but additionally where it is impossible for a *parallel* algorithm to asymptotically achieve a significantly lower cost. Since memory-hard algorithms asymptotically come close to using the most space possible given their running time, and memory is the computationally usable resource general-purpose computers have which is most expensive to reproduce in hardware[8], we believe that, for any given running time on a sequential general-purpose computer, functions which are sequential memory-hard come close to being the most expensive possible functions to compute in hardware.

Indeed, it is surprising to consider the effect of adjusting parameters so that a sequential memory-hard function takes twice as long to compute: As long as there is enough random-access memory to compute the function on a general-purpose system, doubling the time spent asymptotically results in the cost of computing the function in hardware increasing *four-fold*, since the time and required space are both doubled.

[8]On many general-purpose systems, the CPU and motherboard are more expensive than the RAM; but the vast majority of that cost is due to the requirements of general-purpose computation, rapid sequential computation, and support for peripheral devices. The area occupied by *computation* logic on modern CPUs is vanishingly small compared to caches, instruction decoding, out-of-order execution, and I/O.

5. ROMix

The definition of sequential memory-hard functions, while theoretically interesting in its consequences, would not be of any practical value if it turned out that no sequential memory-hard functions exist; to that end, we introduce the class of functions $\mathrm{ROMix}_H : \{0,1\}^k \times \{0 \dots 2^{k/8} - 1\} \rightarrow \{0,1\}^k$ computed as follows[9]:

Algorithm ROMix$_H(B, N)$

Parameters:

	H	A hash function.
	k	Length of output produced by H, in bits.
	Integerify	A bijective function from $\{0,1\}^k$ to $\{0, \dots 2^k - 1\}$.

Input:

	B	Input of length k bits.
	N	Integer work metric, $< 2^{k/8}$

Output:

	B'	Output of length k bits.

Steps:

$X \leftarrow B$
for $i = 0$ to $N - 1$ **do**
  $V_i \leftarrow X$
  $X \leftarrow H(X)$
end for
for $i = 0$ to $N - 1$ **do**
  $j \leftarrow \mathrm{Integerify}(X) \bmod N$
  $X \leftarrow H(X \oplus V_j)$
end for
$B' \leftarrow X$

This algorithm can be thought of as computing a large number of "random" values, and then accessing them "randomly" in order to ensure that they are all stored in Random Access Memory. Before we can prove anything more formally, we need a simple lemma concerning the iterated application of random oracles.

Lemma 1. *Suppose that two algorithms, Algorithm A and Algorithm B exist such that*

(1) *Algorithm A takes as input the integers N, M, and k, a k-bit value B, and an oracle $H : \{0,1\}^k \rightarrow \{0,1\}^k$, and produces a kM-bit output value $A_{N,M,k}(B, H)$; and*
(2) *Algorithm B takes as input the integers N, M, k, and x, with $0 \leq x < N$, and $A_{N,M,k}(B, H)$; and operates on a system which can simultaneously consult M copies of the oracle H in unit time and perform any other computations instantaneously, to compute the value $H^x(B)$.*

Then if the values $H^0(B) \dots H^{N-1}(B)$ are distinct and $N < 2^{k/8}$, Algorithm B operates in expected time at least $\frac{N}{4M+2} - \frac{1}{2}$ for random oracles H, k-bit values B, and integers $x \in \{0 \dots N - 1\}$.

---

[9] We expect that for reasons of performance and simplicity, implementors will restrict N to being a power of 2, in which case the function Integerify can be replaced by reading the first (or last) machine-length word from a k-bit block.

Proof. For notational simplicity, we consider N, M, and k to be fixed and omit them from variable subscripts.

Consider the Process B*, taking input $A(B,H)$, and operating on a machine which can simultaneously consult NM copies of the oracle H in unit time and perform any other computations instantaneously, defined as follows[10]: Execute Algorithm B for each integer $x \in \{0 \ldots N-1\}$ in parallel (using up to M oracles for each value x); after Algorithm B has completed for a value x and has returned the value $H^x(B)$, input that value to an oracle in the following step; finally, if any oracles are unneeded (due to Algorithm B not using all M oracles at some point, due to Algorithm B having finished, or because two or more instances of Algorithm B are inputting the same value into oracles) then input arbitrary unique values to them[11].

Now define $R_i(B,H) \in \{0,1\}^k$ for $i \leq N/M - 1$ to be the set of values input by Process B* to the NM oracles at time i, define $\bar{R}_i(B,H) = R_0(B,H) \cup R_1(B,H) \ldots R_i(B,H)$, and define $\bar{H}(B) = \{H^0(B), \ldots H^{N-1}(B)\}$. Clearly if Algorithm B computes $H^x(B)$ in time t for some B, H, x, then $H^x(B) \in R_i(B,H) \subset \bar{R}_i(B,H)$ for all $i \geq t$. We will proceed by bounding the expected size of $\bar{R}_i(B,H) \cap \bar{H}(B)$ for any process taking a kM bit input and operating on NM oracles.

Let $\bar{R}_{i-1}(B,H)$ and the values of H evaluated thereupon be fixed, and consider the probability, over random B, H, that $H^x(B) \in \bar{R}_i(B,H)$. Trivially, if $H^{x-1}(B) \in \bar{R}_{i-1}(B,H)$, then $P(H^x(B) \in \bar{R}_i(B,H)) \leq 1$ (since the probability of anything is at most 1); but if $H^{x-1}(B) \notin \bar{R}_{i-1}(B,H)$ then the value of H evaluated at $H^{x-1}(B)$ is entirely random; so $P(H^x(B) \in \bar{R}_i(B,H)) = \left|\bar{R}_i(B,H)\right| \cdot 2^{-k} = NM(i+1)2^{-k} \leq N^2 2^{-k}$. Now suppose that out of the $2^{2^k+1-NMi}$ values of (B,H) such that H takes the specified values, $s \cdot 2^{2^k+1-NMi}$ of them (i.e., a proportion s) result share the same value $A(B,H)$. Then the values of $\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|$ for such B, H are at most equal to the $s \cdot 2^{2^k+1-NMi}$ largest values of $\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|$ for permissible H.

However, the Chernoff bound states that for X a sum of independent 0-1 variables, $\mu = E(X)$ and Y a constant greater than μ,

$$P(X > Y) < \exp(Y - \mu + Y(\log \mu - \log Y)),$$

and so for $Y > 1$ we have

$$\begin{aligned} P(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right| - &\left|\bar{R}_{i-1}(B,H) \cap \bar{H}(B)\right| > Y) \\ &< \exp(Y - N^3 2^{-k} + Y(\log(N^3 2^{-k}) - \log(Y))) \\ &< \exp(Y + Y \log(N^3 2^{-k})) \\ &= \left(eN^3 2^k\right)^Y < \left(2^{k/2}\right)^Y \end{aligned}$$

and thus we find that, for (B,H) in the set of $s \cdot 2^{2^k+1-NMi}$ values such that $A(B,H)$ and H evaluated on $\bar{R}_{i-1}(B,H)$ take the correct values,

$$E(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|) < \left|\bar{R}_i(B,H) \cap \bar{H}(B)\right| + \log(s^{-1})/\log(2^{k/2}) + 1,$$

[10] We refer to the *Process* B* instead of the *Algorithm* B* since it neither produces output nor terminates.

[11] Thus for any B, H, Process B* inputs disjoint sets of NM values to the oracles at successive time steps.

where the $+1$ arises as a trivial upper bound on the contribution from the exponentially decreasing probabilities of values $X - Y$ for $X > Y$.

Now we note that $E(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|)$, with expectation taken over *all* values (B,H) is merely the average of the 2^{NMi+Mk} values $E(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|)$ with the expectation taken over (B,H) consistent with a given $A(B,H)$ and values of H at $\bar{R}_{i-1}(B,H)$; and by convexity, the resulting bound is weakest when all of the values s are equal, i.e., $s = 2^{-Mk}$. Consequently, we obtain (with the expectation taken over all (B,H))

$$\begin{aligned} E(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|) &< \left|\bar{R}_i(B,H) \cap \bar{H}(B)\right| + 2M + 1 \\ &< (2M+1) \cdot (i+1). \end{aligned}$$

The result now follows easily: Writing t_x as the expected time taken by Algorithm B to compute $H^x(B)$ and noting that the time it takes to compute $H^x(B)$ is equal to the number of sets $\bar{R}_i(B,H)$ which do not contain $H^x(B)$, we have

$$\begin{aligned} \frac{1}{N}\sum_{x=0}^{N-1} t_x &= \frac{1}{N}\sum_{i=0}^{\infty} N - E(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|) \\ &\geq \frac{1}{N}\sum_{i=0}^{\frac{N}{2M+1}-1} N - E(\left|\bar{R}_i(B,H) \cap \bar{H}(B)\right|) \\ &> \frac{1}{N}\sum_{i=0}^{\frac{N}{2M+1}-1} N - (2M+1)(i+1) \\ &= \frac{N}{4M+2} - \frac{1}{2} \end{aligned}$$

as required.

□

While this proof is rather dense, the idea behind it is quite simple: If the value $H^{x-1}(B)$ has not yet been computed, there is no way to compute $H^x(B)$; and with only kM bits of information stored in $A(B,H)$, any algorithm will be limited to computing $\mathrm{O}(M)$ values $H^x(B)$ from $A(B,H)$ directly and then iterating H to compute the rest. We believe that the "correct" lower bound on the expected running time is in fact $\frac{N}{2M} - \frac{1}{2}$, but this appears difficult to prove.

In spite of being marginally weaker than desirable, this lemma is still sufficient to prove the following theorem:

Theorem 1. *Under the Random Oracle model, the class of functions ROMix$_H$ are sequential memory-hard.*

Proof. The algorithm stated earlier uses $\mathrm{O}(N)$ storage and operates in $\mathrm{O}(N)$ time on a Random Access Machine, so clearly the functions can be computed by a memory-hard algorithm in $T(N) = \mathrm{O}(N)$ operations.

Now suppose that ROMix$_H$ can be computed in $S^*(N) = M(N)$ space. Since H is a random oracle, it is impossible to compute the function without computing each of the values V_j and X in steps 6–9 of the sequential algorithm in turn; but by Lemma 1, it takes at least $\mathrm{O}(N/M(N))$ time to compute each V_j.

Consequently, it takes at least $T^*(N) = \mathrm{O}(N^2/M(N))$ time to compute the function, and thus $S^*(N)T^*(N) = \mathrm{O}(N^2)$ as required. □

6. SMix

While ROMix performs well on a theoretical Random Access Machine, it suffers somewhat on real-world machines. In the real world, random access memory isn't; instead, factors such as caches, prefetching, and virtual memory[12] make small "random" memory accesses far more expensive than accesses to consecutive memory locations.

Existing widely used hash functions produce outputs of up to 512 bits (64 bytes), closely matching the cache line sizes of modern CPUs (typically 32–128 bytes), and the computing time required to hash even a very small amount of data (typically 200–2000 clock cycles on modern CPUs, depending on the hash used) is sufficient that the memory latency cost (typically 100–500 clock cycles) does not dominate the running time of ROMix.

However, as semiconductor technology advances, it is likely that neither of these facts will remain true. Memory latencies, measured in comparison to CPU performance or memory bandwidth, have been steadily increasing for decades, and there is no reason to expect that this will cease — to the contrary, switching delays impose a lower bound of $\Omega(\log N)$ on the latency of accessing a word in an N-byte RAM, while the speed of light imposes a lower bound of $\Omega(\sqrt{N})$ for 2-dimensional circuits. Furthermore, since most applications exhibit significant locality of reference, it is reasonable to expect cache designers to continue to increase cache line sizes in an attempt to trade memory bandwidth for (avoided) memory latency.

In order to avoid having ROMix become latency-limited in the future, it is necessary to apply it to larger hash functions. While we have only proved that ROMix is sequential memory-hard under the Random Oracle model, by considering the structure of the proof we note that the full strength of this model does not appear to be necessary. The critical properties of the hash function required in order for ROMix to be sequential memory-hard appear to be the following[13]:

(1) The outputs of H are uniformly distributed over $\{0,1\}^k$.
(2) It is impossible to iterate H quickly, even given access to many copies of the oracle and precomputation producing a limited-space intermediate.
(3) It is impossible to compute $\text{Integerify}(H(x))$ significantly faster than computing $H(x)$.

Most notably, there is no requirement that the function H have the usual properties of collision and pre-image resistance which are required of cryptographic hashes.

There are also two more criteria required of the hash function in order for ROMix to maximize the cost of a brute-force attack given an upper bound on the amount of computing time taken to compute the function in software:

(4) The ratio of the hash length k to the number of operations required to compute the hash function should be as large as possible.
(5) The hash function should not have significantly more internal parallelism than is available to software implementations.

---

[12] Even if data is stored in RAM, the first access to a page typically incurs a significant cost as the relevant paging tables are consulted.

[13] The first requirement limits the number of values $H^x(B)$ which $A(B, H)$ can uniquely identify; the second requirement ensures that values $H^x(B)$ which are not stored cannot be computed quickly; and the third requirement ensures that each iteration of the loop in lines 6–9 must complete before the next iteration starts.

In light of these, we define the function $\mathrm{BlockMix}_{H,r}$ computed as follows:

Algorithm BlockMix$_{H,r}(B)$

Parameters:

H	A hash function.
r	Block size parameter

Input:

$B_0 \dots B_{2r-1}$	Input vector of $2r$ k-bit blocks

Output:

$B'_0 \dots B'_{2r-1}$	Output vector of $2r$ k-bit blocks.

Steps:

1: $X \leftarrow B_{2r-1}$
2: **for** $i = 0$ to $2r - 1$ **do**
3: $\quad X \leftarrow H(X \oplus B_i)$
4: $\quad Y_i \leftarrow X$
5: **end for**
6: $B' \leftarrow (Y_0, Y_2, \dots Y_{2r-2}, Y_1, Y_3, \dots Y_{2r-1})$

This function clearly satisfies condition (1) if the underlying H is uniformly distributed; it satisfies condition (3) if $\mathrm{Integerify}(B_0 \dots B_{2r-1})$ is defined as a function of B_{2r-1}; and it is clearly optimal according to criteria (4) and (5) compared to any functions constructed out of the same underlying H. We conjecture that BlockMix also satisfies criteria (2), on the basis that the "shuffling" which occurs at step 6 should thwart any attempt to rapidly iterate BlockMix using precomputed values which uniquely identify some but not all of the values B_i; but this does not appear to be easily proven[14].

Given that the performance of BlockMix according to criteria (4) and (5) is exactly the same as the performance of the underlying hash H, BlockMix is best used with a hash which is fast while not possessing excess internal parallelism; based on this, it appears that Bernstein's Salsa20/8 core [11] is the best-performing widely studied cryptographic function available[15]. While Bernstein recommends using the Salsa20 core by adding diagonal constants [13] and uses it in this manner in his Salsa20 cipher and Rumba20 compression functions, we do not believe that this is necessary when the Salsa20 core is being used in ROMix and BlockMix, since the related-input attacks against which they defend are not relevant in this context.

Putting this together, we have the following:

Definition 3. *The function* $\mathit{SMix}_r : \{0,1\}^{1024r} \times \{0 \dots 2^{64} - 1\} \rightarrow \{0,1\}^{1024r}$ *is* $\mathit{SMix}_r(B, N) = \mathit{ROMix}_{\mathit{BlockMix}_{\mathit{Salsa20/8},r}}(B, N)$ *where* $\mathit{Integerify}(B_0 \dots B_{2r-1})$ *is defined as the result of interpreting* B_{2r-1} *as a little-endian integer.*

Theorem 2. *The function* $\mathit{SMix}_r(B, N)$ *can be computed in* $4Nr$ *applications of the Salsa20/8 core using* $1024Nr + \mathrm{O}(r)$ *bits of storage.*

Proof. The above algorithms operate in the required time and space. □

[14] If the shuffling is omitted from BlockMix, it can be rapidly iterated given precomputed values B_0, since the computations would neatly "pipeline".

[15] Bernstein's Chacha [12] appears to have a very slight advantage over Salsa20, but is newer and less widely used, and consequently has been less studied.

7. scrypt

Given a sequential memory-hard "mixing" function MF and a pseudorandom function PRF it is simple to construct a strong key derivation function. We define the class of functions $\mathrm{MFcrypt}_{PRF,MF}(P, S, N, p, dkLen)$ as computed by the following algorithm:

Algorithm MFcrypt$_{H,MF}(P, S, N, p, dkLen)$

Parameters:

PRF	A pseudorandom function.
$hLen$	Length of output produced by PRF, in octets.
MF	A sequential memory-hard function from $\mathbb{Z}_{256}^{MFLen} \times \mathbb{N}$ to $\mathbb{Z}_{256}^{MFLen}$.
$MFLen$	Length of block mixed by MF, in octets.

Intput:

P	Passphrase, an octet string.
S	Salt, an octet string.
N	CPU/memory cost parameter.
p	Parallelization parameter; a positive integer satisfying $p \leq (2^{32} - 1)hLen/MFLen$.
$dkLen$	Intended output length in octets of the derived key; a positive integer satisfying $dkLen \leq (2^{32} - 1)hLen$.

Output:

DK	Derived key, of length $dkLen$ octets.

Steps:

$(B_0 \ldots B_{p-1}) \leftarrow \mathrm{PBKDF2}_{PRF}(P, S, 1, p \cdot MFLen)$
for $i = 0$ to $p - 1$ **do**
  $B_i \leftarrow MF(B_i, N)$
end for
$DK \leftarrow \mathrm{PBKDF2}_{PRF}(P, B_0 \parallel B_1 \parallel \ldots \parallel B_{p-1}, 1, dkLen)$

This algorithm uses PBKDF2 [17] with the pseudorandom function PRF to generate p blocks of length $MFLen$ octets from the provided password and salt; these are independently mixed using the mixing function MF; and the final output is then generated by applying PBKDF2 once again, using the well-mixed blocks as salt[16]. Since, for large N, the calls to MF take asymptotically longer than the calls to PBKDF2, and the blocks B_i produced using PBKDF2 are independent and random, subject to H being a random oracle, we note that if MF is a sequential memory-hard function then MFcrypt is sequential memory-hard under the random oracle model.

We now apply MFcrypt to the mixing function SMix from the previous section and the SHA256 hash function:

Definition 4. *The key derivation function scrypt is defined as*

$$scrypt(P, S, N, r, p, dkLen) = MFcrypt_{HMAC\_SHA256, SMix_r}(P, S, N, p, dkLen)$$

[16]The limits on the size of p and $dkLen$ exist as a result of a corresponding limit on the length of key produced by PBKDF2.

Users of scrypt can tune the parameters N, r, and p according to the amount of memory and computing power available, the latency-bandwidth product of the memory subsystem, and the amount of parallelism desired; at the current time, taking $r = 8$ and $p = 1$ appears to yield good results, but as memory latency and CPU parallelism increase it is likely that the optimum values for both r and p will increase. Note also that since the computations of SMix are independent, a large value of p can be used to increase the computational cost of scrypt without increasing the memory usage; so we can expect scrypt to remain useful even if the growth rates of CPU power and memory capacity diverge.

Conjecture 1. *If it is impossible for a circuit to compute the Salsa20/8 core in less than t time, and it is impossible for a circuit to store x bits of data in less than sx area for any $x \geq 0$, then it is impossible to compute scrypt$(P, S, N, r, p, dkLen)$ in a circuit with an expected amortized area-time product per password of less than $1024N^2r^2pst$.*

Put simply, this conjecture states that combining MFcrypt, ROMix, BlockMix, and the Salsa20/8 core does not expose scrypt to any attacks more powerful than the "generic" algorithms for computing ROMix.

8. Brute-force attack costs

Given a set of key derivation functions, it is natural to ask how much it would cost an attacker to perform a brute-force search over a class of passwords in order to find a particular password given its hash (or, equivalently, given some cryptographic ciphertext which can be used to quickly accept or reject potential password hashes). It is difficult to obtain accurate data concerning the cost of hardware password-cracking circuits — those few organizations which have the resources and inclination to design and fabricate custom circuits for password-cracking tend to be somewhat secretive — and so we must rely instead on estimating the costs of the underlying cryptographic operations in the expectation that the other costs are comparatively negligible. Even given this approximation the amount of information available is limited, since much of the work of implementing cryptographic circuits has been performed by private corporations which have clear financial reasons to restrict access to information about their products to potential customers.

Based on available data concerning DES [1, 4, 5], MD5 [2, 6], Blowfish [14, 21], SHA-256 [3, 7, 20], and Salsa20 [16, 27] cores, we provide the following estimates for the size and performance of cryptographic circuits on a 130 nm process[17]:

- A DES circuit with ≈ 4000 gates of logic can encrypt data at 2000 Mbps.
- An MD5 circuit with ≈ 12000 gates of logic can hash data at 2500 Mbps.
- A SHA256 circuit with ≈ 20000 gates of logic can hash data at 2500 Mbps.
- A Blowfish circuit with ≈ 22000 gates of logic and 4 kiB of SRAM can encrypt data at 1000 Mbps.
- A Salsa20/8 circuit with ≈ 24000 gates of logic can output a keystream at 2000 Mbps.

We also make estimates of the cost of manufacturing integrated circuits on a 130 nm process circa 2002:

---

[17]We use 130 nm as a basis for comparison simply because this is the process technology for which the most information was readily available concerning cryptographic circuits.

- Each gate of random logic requires $\approx 5\ \mu\text{m}^2$ of VLSI area.
- Each bit of SRAM requires $\approx 2.5\ \mu\text{m}^2$ of VLSI area.
- Each bit of DRAM requires $\approx 0.1\ \mu\text{m}^2$ of VLSI area.
- VLSI circuits cost $\approx 0.1\$/\text{mm}^2$.

Using these values, we estimate the cost of computing 9 key derivation functions: the original CRYPT; the MD5 hash (which, although not designed for use as a key derivation function, is nonetheless used as such by many applications); Kamp's MD5-based hash; PBKDF2-HMAC-SHA256 with an iteration count of 86,000; PBKDF2-HMAC-SHA256 with an iteration count of 4,300,000; bcrypt with $cost = 11$; bcrypt with $cost = 16$; scrypt with $(N, r, p) = (2^{14}, 8, 1)$; and scrypt with $(N, r, p) = (2^{20}, 8, 1)$. For the parameterized KDFs the parameters are chosen such that the running time on one core of a 2.5 GHz Intel Core 2 Duo processor[18] is less than 100 ms (for the lower parameters) or less than 5 s (for the higher parameters); we chose these values since 100 ms is a reasonable upper bound on the delay which should be cryptographically imposed on interactive logins, while 5 s is a reasonable amount of time to be spent encrypting or decrypting a sensitive file.

For each key derivation function, we consider six different types of password:

- A random sequence of 6 lower-case letters; e.g., "`sfgroy`".
- A random sequence of 8 lower-case letters; e.g., "`ksuvnwyf`".
- A random sequence of 8 characters selected from the 95 printable 7-bit ASCII characters; e.g., "`6,uh3y[a`".
- A random sequence of 10 characters selected from the 95 printable 7-bit ASCII characters; e.g., "`H.*W8Jz&r3`".
- A 40-character string of text; e.g., "`This is a 40-character string of English`".
- An 80-character string of text; e.g., "`This is an 80-character phrase which you probably won't be able to crack easily.`".

For the strings of text, we estimate entropy following the guidance provided by NIST [23]: The first character is taken to have 4 bits of entropy, the next 7 characters are taken to have 2 bits of entropy each, the following 12 characters are taken to have 1.5 bits of entropy each, and subsequent characters are taken to have 1 bit of entropy each.

In Table 1 we show the estimated costs of "cracking" hashed passwords in dollar-years; or equivalently, the cost of hardware which can find a password in an average time of 1 year (i.e., which would take 2 years to search the complete password space). We caution again that these values are very approximate and reflect only the cost of the cryptographic circuitry with circa 2002 technology: It is quite possible that the costs of other hardware (control circuitry, boards, power supplies) and operating costs (power, cooling) would increase the costs by a factor of 10 above these; and it is equally possible that improvements in semiconductor technology and improved cryptographic circuit designs could each reduce the costs by a factor of 10. Nevertheless, we believe that the estimates presented here are useful for the purpose of comparing different key derivation functions.

It is clear from this table that scrypt is a much more expensive key derivation function to attack than the alternatives: When used for interactive logins, it is 35 times more expensive than bcrypt and 260 times more expensive than PBKDF2;

[18] This processor is also known as "the CPU in the author's laptop".

TABLE 1. Estimated cost of hardware to crack a password in 1 year.

KDF	6 letters	8 letters	8 chars	10 chars	40-char text	80-char text
DES CRYPT	< \$1	< \$1	< \$1	< \$1	< \$1	< \$1
MD5	< \$1	< \$1	< \$1	\$1.1k	\$1	\$1.5T
MD5 CRYPT	< \$1	< \$1	\$130	\$1.1M	\$1.4k	\$1.5 × 10^{15}
PBKDF2 (100 ms)	< \$1	< \$1	\$18k	\$160M	\$200k	\$2.2 × 10^{17}
bcrypt (95 ms)	< \$1	\$4	\$130k	\$1.2B	\$1.5M	\$48B
scrypt (64 ms)	< \$1	\$150	\$4.8M	\$43B	\$52M	\$6 × 10^{19}
PBKDF2 (5.0 s)	< \$1	\$29	\$920k	\$8.3B	\$10M	\$11 × 10^{18}
bcrypt (3.0 s)	< \$1	\$130	\$4.3M	\$39B	\$47M	\$1.5T
scrypt (3.8 s)	\$900	\$610k	\$19B	\$175T	\$210B	\$2.3 × 10^{23}

and when used for file encryption — where, unlike bcrypt and PBKDF2, scrypt uses not only more CPU time but also increases the die area required — scrypt increases its lead to a factor of 4000 over bcrypt and 20000 over PBKDF2. It is also worth noting that while bcrypt is stronger than PBKDF2 for most types of passwords, it falls behind for long passphrases; this results from bcrypt's inability to use more than the first 55 characters of a passphrase[19]. While our estimated costs and NIST's estimates of passphrase entropy suggest that bcrypt's 55-character limitation is not likely to cause problems at the present time, implementors of systems which rely on bcrypt might be well-advised to either work around this limitation (e.g., by "pre-hashing" a passphrase to make it fit into the 55-character limit) or to take steps to prevent users from placing too much password entropy in the 56th and subsequent characters (e.g., by asking users of a website to type their password into an input box which only has space for 55 characters).

9. Conclusions

We have proven that, under the random oracle model, the mixing function ROMix_H is sequential memory-hard; and it appears very likely that the scrypt key derivation function is also sequential memory-hard. Providing that no new attacks on scrypt or its underlying components are found, a brute-force attack on scrypt is many times harder than similar attacks on other key derivation functions; consequently, we recommend that implementors of new cryptographic systems should strongly consider using scrypt.

Finally, we recommend that cryptographic consumers make themselves aware of the strengths of the key derivation functions they are using, and choose passwords accordingly; we suspect that even generally security-conscious users are in many cases not aware how (in)secure their passwords are.

10. Acknowledgements

We thank Arnold G. Reinhold, Daniel J. Bernstein, Graeme Durant, and Paul Kocher for the advice and information they have provided.

[19]This is, however, far better than the original DES-based CRYPT, which only hashed the first 8 bytes of a password and is consequently absurdly cheap to break, regardless of the underlying password distribution.

References

[1] CAST DES data encryption standard core. http://www.cast-inc.com/cores/des/cast_des.pdf.

[2] CAST MD5 hash function core. http://www.cast-inc.com/cores/md5/cast_md5.pdf.

[3] CAST SHA256 secure hash function core. http://www.cast-inc.com/cores/sha-256/cast_sha256.pdf.

[4] DES1 ultra-compact data encryption standard (DES/3DES) core. http://www.ipcores.com/DES1core.htm.

[5] Helion technology datasheet - high performance DES and triple-DES core for asic. http://www.heliontech.com/downloads/des_asic_helioncore.pdf.

[6] Helion technology datasheet - high performance MD5 hash core for asic. http://www.heliontech.com/downloads/md5_asic_helioncore.pdf.

[7] Helion technology datasheet - high performance SHA-256 hash core for asic. http://www.heliontech.com/downloads/sha256_asic_fast_helioncore.pdf.

[8] M. Abadi, T. Mark, A. Lomas, and R. Needham. Strengthening passwords. Technical report, SRC Technical Note, 1997.

[9] C. Bays and S.D. Durham. Improving a poor random number generator. *ACM transactions on mathematical software*, 2(1):59–64, 1976.

[10] D.J. Bernstein. Circuits for integer factorization: a proposal, 2001. http://cr.yp.to/papers.html#nfscircuit.

[11] D.J. Bernstein. The Salsa20 family of stream ciphers, 2007. http://cr.yp.to/papers.html#salsafamily.

[12] D.J. Bernstein. ChaCha, a variant of Salsa20, 2008. http://cr.yp.to/papers.html#chacha.

[13] D.J. Bernstein. Personal communication, 2009.

[14] G. Durant. Personal communication, 2009.

[15] D. Florêncio and C. Herley. A large-scale study of web password habits. In *WWW '07: Proc. of the 16th international World Wide Web conference*, pages 657–666, 2007.

[16] T. Good and M. Benaissa. Hardware results for selected stream cipher candidates. In *Proc. of The State of the Art of Stream Ciphers*, 2007.

[17] B. Kaliski. PKCS #5: Password-based cryptography specification version 2.0. RFC 2898, 2000.

[18] P.-H. Kamp. MD5 crypt. FreeBSD 2.0, 1994. http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/lib/libcrypt/crypt.c?rev=1.2.

[19] J. Kelsey, B. Schneier, C. Hall, and D. Wagner. Secure applications of low-entropy keys. In *ISW '97: Proc. of the first international workshop on information security*, pages 121–134, 1998.

[20] Y.K. Lee, H. Chan, and I. Verbauwhede. Iteration bound analysis and throughput optimum architecture of SHA-256 (384, 512) for hardware implementations. In *Workshop on Information Security Applications 2007*, LNCS 4867, 2008.

[21] M.C.-J. Lin and Y.-L. Lin. A VLSI implementation of the blowfish encryption/decryption algorithm. In *ASP-DAC '00: Proceedings of the 2000 conference on Asia South Pacific design automation*, 2000.

[22] R.H. Morris and K. Thompson. UNIX password security. *Communications of the ACM*, 22(11), 1979.

[23] National Institute of Standards and Technology. Electronic authentication guideline. NIST Special Publication 800-63, 2006.

[24] N. Provos and D. Mazières. A future-adaptable password scheme. In *Proc. of the FREENIX track: 1999 USENIX annual technical conference*, 1999.

[25] A.G. Reinhold. HEKS: A family of key stretching algorithms, 1999. http://world.std.com/~reinhold/HEKSproposal.html.

[26] A.G. Reinhold. Personal communication, 2009.

[27] J. Yan and H.M. Heys. Hardware implementation of the Salsa20 and Phelix stream ciphers. In *Proc. of the IEEE Canadian Conference on Electrical and Computer Engineering*, 2007.

Appendix A. Availability

Source code for scrypt, including reference and optimized implementations in C, and a demonstration file-encryption utility are available for download and use under the 2-clause BSD license from `http://www.tarsnap.com/scrypt/`.

Appendix B. Test vectors

For reference purposes, we provide the following test vectors for scrypt, where the password and salt strings are passed as sequences of ASCII bytes without a terminating NUL:

scrypt("", "", 16, 1, 1, 64) =

```
77 d6 57 62 38 65 7b 20 3b 19 ca 42 c1 8a 04 97
f1 6b 48 44 e3 07 4a e8 df df fa 3f ed e2 14 42
fc d0 06 9d ed 09 48 f8 32 6a 75 3a 0f c8 1f 17
e8 d3 e0 fb 2e 0d 36 28 cf 35 e2 0c 38 d1 89 06
```

scrypt("password", "NaCl", 1024, 8, 16, 64) =

```
fd ba be 1c 9d 34 72 00 78 56 e7 19 0d 01 e9 fe
7c 6a d7 cb c8 23 78 30 e7 73 76 63 4b 37 31 62
2e af 30 d9 2e 22 a3 88 6f f1 09 27 9d 98 30 da
c7 27 af b9 4a 83 ee 6d 83 60 cb df a2 cc 06 40
```

scrypt("pleaseletmein", "SodiumChloride", 16384, 8, 1, 64) =

```
70 23 bd cb 3a fd 73 48 46 1c 06 cd 81 fd 38 eb
fd a8 fb ba 90 4f 8e 3e a9 b5 43 f6 54 5d a1 f2
d5 43 29 55 61 3f 0f cf 62 d4 97 05 24 2a 9a f9
e6 1e 85 dc 0d 65 1e 40 df cf 01 7b 45 57 58 87
```

scrypt("pleaseletmein", "SodiumChloride", 1048576, 8, 1, 64) =

```
21 01 cb 9b 6a 51 1a ae ad db be 09 cf 70 f8 81
ec 56 8d 57 4a 2f fd 4d ab e5 ee 98 20 ad aa 47
8e 56 fd 8f 4b a5 d0 9f fa 1c 6d 92 7c 40 f4 c3
37 30 40 49 e8 a9 52 fb cb f4 5c 6f a7 7a 41 a4
```