What output size resists collisions in a xor of independent expansions?

Daniel J. Bernstein

Department of Mathematics, Statistics, and Computer Science (MC 249)
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to

Abstract. Bellare and Micciancio proposed compressing $(m_1, m_2, \ldots)$ to $f_1(m_1) \oplus f_2(m_2) \oplus \cdots$. Collisions are easy to find for long messages but are much more difficult to find for short messages. Exactly how secure is the 4-xor compression function $(m_1, m_2, m_3, m_4) \mapsto f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4)$, with an output size of $4b$ bits? This paper analyzes, under constraints on machine cost and computation time, the chance of finding $4b$-bit collisions using an improved version of Wagner's generalized-birthday algorithm. In particular, as the machine cost grows past $2^{2b/3}$, the price-performance ratio of this paper's attack drops below 2^{2b}, eventually reaching a limit of $2^{4b/3}$. This paper also proposes the Rumba20 compression function, reusing large components of the Salsa20 stream cipher as a specific choice of functions f_1, f_2, f_3, f_4.

Keywords: hash functions, compression functions, collision resistance, cryptanalysis, generalized birthday attacks, parallelization, compression-function design, stream-cipher synergy

1 Introduction

How hard is it to find collisions in the compression function $(m_1, m_2, m_3, m_4) \mapsto f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4)$?

Assume, for concreteness, that each f_i expands a 384-bit input to a 512-bit output. This compression function then maps a 1536-bit input to a 512-bit output, effectively eliminating 256 bits for each f_i evaluation. More generally, if each f_i expands a $3b$-bit input to a $4b$-bit output, then the compression function maps a $12b$-bit input to a $4b$-bit output. How large does b have to be for this function to resist collisions?

Previous answers. The standard answer is that Wagner's generalized-birthday algorithm in [15] takes time 2^b, improving on the time 2^{2b} taken by traditional $4b$-bit collision search. But this answer has several glaring deficiencies:

---

[*] Date of this document: 2007.05.03.

[*] Permanent ID of this document: dee774697b66f19a00071db1b0666cab.

[*] This work was carried out while the author was visiting Technische Universiteit Eindhoven.

- The standard answer fails to account for limits on the attacker's time. The generic success chance of traditional collision search is well known to drop quadratically as the time spent drops; how badly does the generic success chance of Wagner's algorithm drop as the time spent drops? ("Generic success chance" here means the average success chance for *all* functions; the success chance for a particular choice of function could be different.)
- The standard answer fails to account for limits on machine cost. Wagner's algorithm needs a terrifyingly large machine with 2^b blocks of memory—for example, 2^{128} blocks for $b = 128$. For comparison, traditional collision search can be carried out by a tiny circuit, only slightly larger than a circuit to compute the compression function. How badly does the generic success chance of Wagner's algorithm drop with the machine cost?
- The standard answer relies on the assumption that each memory access in Wagner's algorithm takes constant time. In fact, speed-of-light delays force each storage access to take time $2^{b/2}$, so the 2^b serial storage accesses in Wagner's algorithm take time $2^{3b/2}$.

The bottom line is that Wagner's algorithm has a price-performance ratio on the scale of $2^{5b/2}$, considerably *worse* than the price-performance ratio of traditional collision search, despite being advertised as a faster algorithm. An attacker with any particular hardware budget and time budget will find more collisions with traditional collision search than with Wagner's algorithm.

Wagner in [15, Section 5, "Open Problems," under "Memory and communication complexity"] asked whether his algorithm's memory requirements could be reduced; asked whether the algorithm could be "parallelized effectively without enormous communication complexity"; and advised designers "to assume that such algorithmic improvements may be forthcoming."

Contributions of this paper. This paper presents new upper bounds for the price-performance ratio of generic collision-finding algorithms for $f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4)$. In particular, this paper improves Wagner's algorithm, drastically reducing the time to $2^{4b/9}$ while reducing the machine size from 2^b to $2^{8b/9}$.

More generally, for each c between 0 and $8b/9$, this paper presents an attack taking time 2^{4b-4c} on a circuit of size 2^c. The price-performance ratio of the circuit is 2^{4b-3c}, rising from $2^{4b/3}$ for $c = 8b/9$ to 2^{2b} for $c = 2b/3$ and then higher as c drops. For $c < 2b/3$ it is better to use the parallel-collision-search circuit of van Oorschot and Wiener in [13], taking time approximately 2^{2b-c} on a circuit of size 2^c.

Even more generally, for each c between 0 and $8b/9$ and for each t between $c/2$ and $4b - 4c$, this paper presents an attack taking time 2^t on a circuit of size 2^c and having generic success probability approximately $2^{t+4c-4b}$. For $t > 2c$ it is better to use the parallel-collision-search circuit of van Oorschot and Wiener, taking time 2^t on a circuit of size 2^c and having generic success probability approximately $2^{2t+2c-4b}$.

Section 2 presents the new attack. Section 3 discusses the history of the compression function $(m_1, m_2, m_3, m_4) \mapsto f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4)$ and

the motivation for using this function; it also presents the Rumba20 compression function, with a specific choice of f_1, f_2, f_3, f_4, reusing large components of the Salsa20 stream cipher.

Open questions. The new success probability $2^{t+4c-4b}$ increases linearly with the time spent but *quartically* with the circuit size (up to size $2^{8b/9}$). For comparison, the van Oorschot-Wiener success probability $2^{2t+2c-4b}$ increases quadratically with time and quadratically with circuit size. What other functions of t, c can be achieved? Can one achieve $2t + 3c - 4b$, for example, or $2t + 4c - 4b$, or $4t + c - 4b$?

The only obvious limit is $2^{8t+8c-4b}$. A circuit of size 2^c cannot generate more than 2^{t+c} values $f_i(m_i)$ in time 2^t; there exist at most 2^{4t+4c} combinations $f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4)$; there exists a collision among those combinations with probability at most $2^{8t+8c-4b}$. Is there any better upper bound? The Brent-Kung theorem [6, Theorem 3.1] (predating [16, Theorem 1] by more than twenty years) produces better-than-information-theoretic bounds on the price-performance ratio of broadcast computations such as sorting; to what extent can collision-finding algorithms avoid sorting?

Generalizations. One can ask the same questions about, e.g., $f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4) \oplus f_5(m_5) \oplus f_6(m_6) \oplus f_7(m_7) \oplus f_8(m_8)$. This paper's analysis readily generalizes to a larger (or smaller) number of inputs. But this 8-xor compression function is not much faster than the 4-xor compression function—it eliminates $2.5b$ bits for each f_i evaluation, compared to $2b$ bits—and the gain in compression speed seems to be offset by the gain in attack speed.

2 The attack

This section reviews Wagner's algorithm to find collisions in $(m_1, m_2, m_3, m_4) \mapsto f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4)$. This section then adapts the algorithm to handle constraints on machine cost, to take advantage of parallel processing, and to take advantage of low-cost precomputation. This section concludes by evaluating and improving constant factors in the cost of the algorithm.

Review of Wagner's algorithm. Choose 2^b different values of m_1 and 2^b different values of m_2. There are 2^{2b} pairs (m_1, m_2), and on average (generically) there are 2^b pairs having the first b bits of $f_1(m_1) \oplus f_2(m_2)$ equal to 0. Find those pairs as follows: compute the 2^b values $(f_1(m_1), m_1)$ and sort them into lexicographic order; compute the 2^b values $(f_2(m_2), m_2)$ and sort them into lexicographic order; merge the sorted lists to find all pairs (m_1, m_2) for which the first b bits of $f_1(m_1)$ agree with the first b bits of $f_2(m_2)$.

Wagner states that the sorting takes "$O(n \log n)$ time" where $n = 2^b$. Presumably "$O(n \log n)$" is meant to refer to heap sort or another standard comparison-based sorting algorithm that sorts n items using $O(n \log n)$ comparisons and $O(n \log n)$ memory accesses.

One can object that a comparison of b-bit strings actually takes time proportional to b, not constant time, so heap sort uses $O(2^b b^2)$ bit comparisons, not

merely the claimed $O(2^b b)$; one can, on the other hand, replace heap sort with radix sort, which eliminates the $\log n$ factor and uses only $O(2^b b)$ bit comparisons. Similarly, and more importantly, one can object that memory accesses do not take constant time; one can, on the other hand, choose a sorting algorithm with much smaller communication costs, as discussed below.

After finding 2^b vectors $(f_1(m_1) \oplus f_2(m_2), m_1, m_2)$ with first b bits equalling 0, use the same idea to find 2^b vectors $(f_3(m_3) \oplus f_4(m_4), m_3, m_4)$ with first b bits equalling 0. Sort and merge these lists of vectors to find, on average, 2^b vectors $(f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4), m_1, m_2, m_3, m_4)$ with first $2b$ bits equalling 0.

Finally, sort this list of vectors to find any collisions in the $4b$-bit strings $f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4)$. The average number of collisions found is $(1/2)2^{4b}(2^{4b} - 1)2^{-8b} = 1/2 - 1/2^{4b+1}$.

One can object that there is no reason for the chance of finding at least one collision to be as large as the average number of collisions found; but a more detailed analysis shows that the gap is negligible for large b. Limiting the intermediate lists to exactly 2^b vectors also makes very little difference in the success probability of the algorithm.

Handling constraints on machine size. Wagner's algorithm needs $\Theta(2^b b)$ bits of memory to store $\Theta(2^b)$ vectors, each having $\Theta(b)$ bits. What if the attacker cannot afford to pay for $\Theta(2^b b)$ bits of memory?

Let's say the attacker can afford to store only 2^c vectors $(f_1(m_1), m_1)$ rather than 2^b vectors, and only 2^c vectors $(f_2(m_2), m_2)$ rather than 2^b vectors. Here c is a parameter that will be reflected in the final machine cost.

The attacker now finds, on average, 2^{2c-b} vectors $(f_1(m_1) \oplus f_2(m_2), m_1, m_2)$ where the first b bits are 0. Together with 2^{2c-b} similar vectors $(f_3(m_3) \oplus f_4(m_4), m_3, m_4)$ the attacker finds 2^{4c-3b} vectors $(f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4), m_1, m_2, m_3, m_4)$ where the first $2b$ bits are 0. The expected number of collisions is approximately 2^{8c-8b}.

Improvement: Increase the number of vectors $(f_1(m_1) \oplus f_2(m_2), m_1, m_2)$ up to the machine capacity 2^c, by requiring only the first c bits to be 0; similarly increase the number of vectors $(f_3(m_3) \oplus f_4(m_4), m_3, m_4)$. The number of these vectors is reflected quadratically in the number of 4-xor vectors, and quartically in the final number of collisions, outweighing the loss of $b - c$ bits. Similarly, increase the number of vectors $(f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4), m_1, m_2, m_3, m_4)$ up to the machine capacity 2^c. The revised attack is as follows:

- Generate and sort 2^c vectors $(f_1(m_1), m_1)$.
- Generate and sort 2^c vectors $(f_2(m_2), m_2)$.
- Merge to find 2^c vectors $(f_1(m_1) \oplus f_2(m_2), m_1, m_2)$ having first c bits equal to 0.
- Similarly find 2^c vectors $(f_3(m_3) \oplus f_4(m_4), m_3, m_4)$ having first c bits equal to 0.
- Sort and merge again to find 2^c vectors $(f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4), m_1, m_2, m_3, m_4)$ having first $2c$ bits equal to 0.
- Sort again to find, on average, about $(1/2)2^{2c-(4b-2c)} = 2^{4c-4b-1}$ collisions.

One can also view the revised attack as follows: truncate each f_i to $4c$ bits; use the original attack to search for collisions in the $4c$-bit strings $f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4)$; hope that the collisions are actually collisions in the original $4b$-bit strings.

Parallelization. As mentioned in Section 1, a realistic model of computation cannot support constant-latency random access to 2^c bits of memory as c grows; one needs at least $2^{c/2}$ time to reach a typical position in a 2-dimensional circuit of size 2^c. A sorting algorithm that issues 2^c serial memory accesses ends up taking time proportional to at least $2^{3c/2}$.

Parallelism offers dramatic improvements. A circuit of size 2^c can handle a pipeline of $2^{c/2}$ parallel memory accesses from a single CPU; some sorting algorithms can take advantage of this, reducing the sorting time from roughly $2^{3c/2}$ to roughly 2^c. Furthermore, a realistic circuit of size 2^c can have roughly 2^c tiny processors acting in parallel; some sorting algorithms can take advantage of this, reducing the sorting time from roughly 2^c to roughly $2^{c/2}$.

Specifically, Schimmler's algorithm in [10] uses an $n \times n$ mesh of n^2 small processors to sort n^2 small objects in approximately $8n$ steps. Each processor has storage for one object, a small amount of comparison circuitry, and wires connecting it to the four adjacent processors. In each step, each processor performs a compare-exchange with an adjacent processor, sorting the two objects in the two processors in an order specified by the algorithm. An alternative to Schimmler's algorithm is the Schnorr-Shamir algorithm in [11], which uses a more complicated order of operations but reduces 8 to approximately 3.

In particular, one can sort 2^c vectors $(f_1(m_1), m_1, 1)$ together with 2^c vectors $(f_2(m_2), m_2, 2)$ by applying Schimmler's algorithm with $n = 2^{(c+1)/2}$. The algorithm uses 2^{c+1} small processors to sort these 2^{c+1} objects in approximately $2^{(c+7)/2}$ compare-exchange steps. Similar comments apply to the other sorting steps required in Wagner's algorithm.

The algorithm also needs 2^c evaluations of f_1, 2^c evaluations of f_2, etc., but 2^c (or fewer) parallel processors can handle these evaluations at negligible cost for any reasonable choice of f_1, f_2, f_3, f_4.

Summary: This parallelized attack uses time on the scale of $2^{c/2}$; uses a machine whose size is on the scale of 2^c; and produces on the scale of 2^{4c-4b} collisions.

Precomputation. There is an imbalance in the above parallelized attack. The computation of 2^c values of $f_1(m_1)$ takes very little time—the same time as computing one value—because it is parallelized perfectly across 2^c small processors. The sorting of those values takes much more time, the time for roughly $2^{c/2}$ compare-exchange steps.

Improvement: Spend more time searching for useful values of m_1. For example, rather than taking the first m_1 that comes to mind, each processor can try $2^{c/2}$ values of m_1, choosing the smallest $f_1(m_1)$ in reverse lexicographic order—typically one where the last $c/2$ bits are 0. Similarly spend more time searching for useful values of m_2, m_3, m_4.

This improvement increases the final number of collisions to the scale of $2^{9c/2-4b}$. It increases the time for the attack to roughly $2^{c/2}$ compare-exchange steps and roughly $2^{c/2}$ evaluations of f_i; still on the scale of $2^{c/2}$ overall, when evaluation of f_i is reasonably fast.

By repeating the same attack $2^{t-c/2}$ times one increases the time to the scale of 2^t and increases the number of collisions to the scale of $2^{t+4c-4b}$, as advertised in Section 1.

Constant-factor improvements. Assume for definiteness that c is odd, and that there are 2^{c+1} processors in a $2^{(c+1)/2} \times 2^{(c+1)/2}$ mesh. Also assume that $9c \leq 8b$.

Half of the processors compute $f_1(m_1)$ for various m_1's, while the other half of the processors compute $f_2(m_2)$ for various m_2's. Specifically, each processor puts a c-bit processor ID into m_i (the ID and i determining the processor), varies the next $(c+1)/2$ bits of m_i, sets the remaining m_i bits to 0, and computes the $2^{(c+1)/2}$ values $f_i(m_i)$, remembering the m_i whose $f_i(m_i)$ value is smallest in reverse lexicographic order. The $f_i(m_i)$ values can be safely compressed to their last c bits; other bits will be needed later but will also be recomputed later, taking negligible extra time.

The processors then sort into lexicographic order the 2^{c+1} vectors

$$(\text{first } c \text{ bits of } f_i(m_i), m_i, i).$$

Each vector can be stored and communicated as $(5c+3)/2$ bits: c bits of $f_i(m_i)$, $(3c+1)/2$ variable bits of m_i, and 1 bit for i. Schimmler's algorithm performs this sorting with $2^{(c+7)/2}$ compare-exchange steps.

All pairs (m_1, m_2) having the same first c bits of $f_1(m_1)$ and $f_2(m_2)$ have now been brought very close together. A negligible number of local communication steps finds all of the pairs (discarding extras if there are more than 2^c) and spreads them among the 2^{c+1} processors. Each pair is stored as one m_i per processor, i.e., $(3c+1)/2$ bits per processor.

The processors now repeat with f_3 and f_4, using another $2^{(c+1)/2}$ evaluations of f_i and another $2^{(c+7)/2}$ compare-exchange steps for Schimmler's algorithm. Each processor needs $(5c+3)/2$ bits to store the vectors being sorted, and $(3c+1)/2$ bits for the saved pairs from the previous vectors, totalling $4c+2$ bits.

The processors then sort the 2^{c+1} vectors

$$(\text{next } c-1 \text{ bits of } f_i(m_i) \oplus f_{i+1}(m_{i+1}), m_i, m_{i+1}, i)$$

for $i \in \{1, 3\}$, using another $2^{(c+7)/2}$ compare-exchange steps. Each vector is stored in $4c+1$ bits.

All (m_1, m_2) and (m_3, m_4) colliding in the next $c-1$ bits of $f_1(m_1) \oplus f_2(m_2)$ and the next $c-1$ bits of $f_3(m_3) \oplus f_4(m_4)$ have now been brought very close together. As before, a negligible amount of extra work finds all of these collisions (discarding extras if there are more than 2^{c+1}) and spreads them among the 2^{c+1} processors.

The processors replace each (m_1, m_2, m_3, m_4) with the next $4c+1$ bits of $f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4)$. If $6c > 4b$ then only $4b-2c+1$ bits are

available and the extra $6c - 4b$ bits can be profitably occupied by some bits of (m_1, m_2, m_3, m_4). The processors then find collisions in these $4c + 1$ bits, using another $2^{(c+7)/2}$ compare-exchange steps.

If a collision occurs, the processors find the relevant m_1 etc. by recomputing the possibilities for $f_1(m_1)$ etc.; this occurs rarely (when c is not very close to $8b/9$), so recomputation is better than spending memory on remembering m_1 etc. Finally, the processors check whether there is a collision in *all* of the bits of $f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4)$.

The total time for this attack is dominated by $2^{(c+3)/2}$ evaluations of f_i and $2^{(c+11)/2}$ compare-exchange steps on $(4c + 1)$-bit strings. Each processor needs four $(4c + 1)$-bit links to adjacent processors, $4c + 2$ bits of storage, a similar amount of comparison circuitry, and a circuit that can compute two of the f_i's. The chance of finding a collision is approximately $2^{(9c+1)/2-4b}$.

Repeating the same computation $2^{t-(c+1)/2}$ times raises the collision-finding chance to approximately $2^{t+4c-4b}$, raises the number of evaluations of f_i to 2^{t+1}, and raises the number of compare-exchange steps to 2^{t+5}. Of course, the relative number of f_i evaluations and compare-exchange steps can and should be adjusted to account for the relative costs of these operations.

3 History, motivation, etc.

Incrementality. Bellare and Micciancio in [3] proposed the general structure $(m_1, m_2, \ldots) \mapsto f_1(m_1) + f_2(m_2) + \cdots$ for a collision-resistant function allowing "incrementality," i.e., fast adjustment of the function value if m_i changes to m_i'. They considered several choices of group operations $+$:

- "XHASH": xor, i.e., addition of vectors modulo 2, i.e., addition in $(\mathbf{Z}/2)^k$.
- "AdHASH": addition modulo a large integer m, i.e., addition in $\mathbf{Z}/m$.
- "LtHASH": addition of vectors modulo m, i.e., addition in $(\mathbf{Z}/m)^k$.
- "MuHASH": multiplication modulo a prime.

Bellare and Micciancio rejected xor ("the most tempting choice, XOR, doesn't work") because of the following easy attack for *very long* inputs: consider the $4b$-bit vectors $f_1(m_1') - f_1(m_1)$, $f_2(m_2') - f_2(m_2)$, etc.; use linear algebra to find a linear dependency modulo 2 between these vectors, i.e., to find $\delta_1, \delta_2, \ldots \in \{0, 1\}$, not all zero, such that $\delta_1(f_1(m_1') - f_1(m_1)) + \delta_2(f_2(m_2') - f_2(m_2)) + \cdots = 0$; observe that $f_1(m_1 + \delta_1(m_1' - m_1)) + f_2(m_2 + \delta_2(m_2' - m_2)) + \cdots$ collides with $f_1(m_1) + f_2(m_2) + \cdots$. But this attack breaks down for short inputs: the chance of a linear dependency among 4 vectors is only about 2^{4-4b}.

Wagner then introduced his algorithm, and in [15, page 14] stated that the algorithm could find long-message collisions in AdHash

> with complexity $O(2^{2\sqrt{\lg m}})$. ... As a consequence of this attack, we will need to ensure that $m \gg 2^{1600}$ if we want 80-bit security. The need for such a large modulus may reduce or negate the performance advantages of AdHash.

But the speed of the attack is not nearly as impressive for short messages, such as the 4-block messages considered in this paper.

Of course, limiting the input length to 4 blocks also makes the incrementality of the function less impressive, but updates can still be 4 times faster than they would have been otherwise.

Reusing ciphers. A stream cipher can be viewed as a function expanding its input (a key and a nonce) into an output stream. Many stream ciphers have large setup costs and aim only at efficiency for long outputs, but some stream ciphers have much smaller setup costs and can be used for very short outputs. At an extreme is the Salsa20 stream cipher that I introduced in [5], which allows random access to blocks of the output stream: Salsa20 expands a 256-bit key, a 64-bit nonce, and a 64-bit block counter into a 512-bit block. Classic "filter generator" stream ciphers also allow fast random access.

It is natural to consider ways to reuse these expansion functions inside hash functions. Reuse allows some sharing of software, some sharing of precious hardware resources, and some sharing of cryptanalysis. The Bellare-Micciancio structure is one of the simplest imaginable approaches, with the interesting feature of allowing highly parallel computation given adequate hardware resources.

Similarly, block ciphers can be viewed as expansion functions: for example, counter-mode AES expands a 128-bit key k and a 128-bit nonce n into the 384-bit string $\mathrm{AES}(k,n), \mathrm{AES}(k,n+1), \mathrm{AES}(k,n+2)$. There is a long history of reuse of block ciphers inside hash functions; see, e.g., [8]. But the constraint of small-block invertibility seems to interfere with attempts to achieve security at high speed. The fastest unbroken block ciphers are considerably slower than the fastest unbroken stream ciphers.

Do ciphers and compression functions have the same security goals? Certainly not. For example, weakness of a negligible fraction of keys has no relevance to stream-cipher security but often allows easy collisions when the keys are viewed as compression-function inputs. One can nevertheless recognize considerable overlap between, e.g., differential cryptanalysis of a stream cipher and differential cryptanalysis of a compression function. The community saves time when this cryptanalytic effort is focused on *one* function.

The Rumba20 compression function. Here is a 1536-bit-to-512-bit compression function Rumba20 obtained by feeding Salsa20 to the above design strategy.

Salsa20 packs its 384-bit input and 128 bits of "diagonal constants" into a 512-bit block, which is then transformed to produce the 512-bit Salsa20 output. The diagonal constants can easily be tweaked to produce four 384-bit-to-512-bit functions f_1, f_2, f_3, f_4:

- f_1 uses diagonal constants `firstRumba20bloc` in ASCII.
- f_2 uses diagonal constants `secondRumba20blo` in ASCII.
- f_3 uses diagonal constants `thirdRumba20bloc` in ASCII.
- f_4 uses diagonal constants `fourthRumba20blo` in ASCII.

Rumba20 compresses a 1536-bit string (m_1, m_2, m_3, m_4) to the 512-bit string $f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4)$, eliminating 1024 bits.

Rumba20 will take about twice as many cycles per eliminated byte as Salsa20 takes per encrypted byte. The eSTREAM project has measured Salsa20 as taking 4.36 cycles per byte on a PowerPC G4 7410, 7.83 cycles per byte on an Athlon 64 X2, 11.89 cycles per byte on a Pentium M, and 16.96 cycles per byte on a Pentium 4 f29, so Rumba20 will be competitive in speed with SHA-256. Detailed comparison will of course require an implementation.

This paper's attack improves the price-performance ratio of Rumba20 collision search if the attacker has more than 2^{85} small processors, and reaches a limiting price-performance ratio on the scale of 2^{171} when the attacker reaches 2^{114} small processors. These computations obviously will not be carried out in the foreseeable future, and would have to be dramatically improved before they become a threat.

Can one do better with non-generic attacks that somehow take advantage of Salsa20? One approach is to find many Salsa20 inputs whose outputs begin with many 0's, or that are confined to some other linear subspace. The Rumba20 output will then be confined to the same subspace, speeding up the attack in this paper. The problem is that finding the inputs seems difficult.

Salsa20, like MD5 and AES and many other cryptographic functions, has a multiple-round structure. Existing cryptanalysis of the Salsa20 stream cipher has focused on reduced-round variants, using fewer than half of Salsa20's 20 rounds—and suggesting that Salsa20 can be made twice as fast with no loss in security. Presumably 20 rounds are also overkill for Rumba20's collision resistance, but it is not obvious exactly how many rounds are required. I look forward to seeing independent cryptanalysis of reduced-round variants of Rumba20.

Going beyond collision resistance. Saarinen in [9] criticized the "VSH" compression function for (1) not hiding its input very well—r unknown bits of a preimage could be found in time $2^{r/2}$, and sometimes much less, rather than the hoped-for 2^r; (2) not allowing truncation—collisions in an r-bit truncation of VSH could be found in time much less than $2^{r/2}$; and (3) in general not hiding its algebraic structure.

Similar criticisms could be leveled against Rumba20, and against any other compression function of the form $(m_1, m_2, m_3, m_4) \mapsto f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4)$.

The obvious response is that, in applications needing more than collision resistance (and perhaps in all applications), the compression-function output should be fed through an output filter before it is given to the application.

References

1. — (no editor), *Proceedings of the 18th annual ACM symposium on theory of computing*, Association for Computing Machinery, New York, 1986. ISBN 0–89791–193–8. See [11].
2. Rana Barua, Tanja Lange (editors), *Progress in Cryptology—INDOCRYPT 2006, 7th International Conference on Cryptology in India, Kolkata, India, December 11–13, 2006, Proceedings*, Lecture Notes in Computer Science, 4329, Springer, 2006. ISBN 3–540–49767–6. See [9].

3. Mihir Bellare, Daniele Micciancio, *A new paradigm for collision-free hashing: incrementality at reduced cost* (1996); see also newer version [4]. URL: `http://www-cse.ucsd.edu/~mihir/papers/incremental.html`. Citations in this document: §3.
4. Mihir Bellare, Daniele Micciancio, *A new paradigm for collision-free hashing: incrementality at reduced cost*, in [7] (1997), 163–192; see also older version [3].
5. Daniel J. Bernstein, *Salsa20*, eSTREAM, ECRYPT Stream Cipher Project, Report 2005/025 (2005). URL: `http://www.ecrypt.eu.org/stream`. Citations in this document: §3.
6. Richard P. Brent, H. T. Kung, *The area-time complexity of binary multiplication*, Journal of the ACM **28**, 521–534. URL: `http://wwwmaths.anu.edu.au/~brent/pub/pub055.html`. Citations in this document: §1.
7. Walter Fumy (editor), *Advances in cryptology: EUROCRYPT '97*, Lecture Notes in Computer Science, 1233, Springer-Verlag, Berlin, 1997. ISBN 3–540–62975–0. See [4].
8. Bart Preneel, René Govaerts, Joos Vandewalle, *Hash functions based on block ciphers: a synthetic approach*, in [12] (1994), 368–378. Citations in this document: §3.
9. Markku-Juhani O. Saarinen, *Security of VSH in the real world*, in [2] (2006), 95–103. URL: `http://eprint.iacr.org/2006/103`. Citations in this document: §3.
10. Manfred Schimmler, *Fast sorting on the instruction systolic array*, report 8709, Christian-Albrechts-Universität Kiel, 1987. Citations in this document: §2.
11. Claus P. Schnorr, Adi Shamir, *An optimal sorting algorithm for mesh-connected computers*, in [1] (1986), 255–261. Citations in this document: §2.
12. Douglas R. Stinson (editor), *Advances in cryptology—CRYPTO '93: 13th annual international cryptology conference, Santa Barbara, California, USA, August 22–26, 1993, proceedings*, Lecture Notes in Computer Science, 773, Springer-Verlag, Berlin, 1994. ISBN 3–540–57766–1, 0–387–57766–1. MR 95b:94002. See [8].
13. Paul C. van Oorschot, Michael Wiener, *Parallel collision search with cryptanalytic applications*, Journal of Cryptology **12** (1999), 1–28. ISSN 0933–2790. URL: `http://members.rogers.com/paulv/papers/pubs.html`. Citations in this document: §1.
14. David Wagner, *A generalized birthday problem (extended abstract)*, in [17] (2002), 288–303; see also newer version [15]. URL: `http://www.cs.berkeley.edu/~daw/papers/genbday.html`.
15. David Wagner, *A generalized birthday problem (extended abstract) (long version)* (2002); see also older version [14]. URL: `http://www.cs.berkeley.edu/~daw/papers/genbday.html`. Citations in this document: §1, §1, §3.
16. Michael J. Wiener, *The full cost of cryptanalytic attacks*, Journal of Cryptology **17** (2004), 105–124. ISSN 0933–2790. URL: `http://www3.sympatico.ca/wienerfamily/Michael/`. Citations in this document: §1.
17. Moti Yung (editor), *Advances in cryptology—CRYPTO 2002: 22nd annual international cryptology conference, Santa Barbara, California, USA, August 2002, proceedings*, Lecture Notes in Computer Science, 2442, Springer-Verlag, Berlin, 2002. ISBN 3–540–44050–X. See [14].