Better price-performance ratios for generalized birthday attacks

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. Fix i and k with $k = 2^{i-1}$. This paper presents a generalized-birthday attack that uses a machine of size $2^{2B/(2i+1)}$ for time $2^{B/(2i+1)}$ to find $(m_1, \ldots, m_k)$ such that $f_1(m_1) + \cdots + f_k(m_k) \bmod 2^B = 0$. The exponents $2/(2i+1)$ and $1/(2i+1)$ are smaller than the exponents for Wagner's original generalized-birthday attack. The improved attack also allows a linear tradeoff between time and success probability, and an ith-power tradeoff between machine size and success probability.

1 Introduction

Fix $k \geq 2$. Let $f_1, f_2, f_3, \ldots, f_k$ be easy-to-compute functions producing B-bit outputs. How difficult is it to find a vector $(m_1, m_2, m_3, \ldots, m_k)$ such that $f_1(m_1) + f_2(m_2) + f_3(m_3) + \cdots + f_k(m_k) \bmod 2^B = 0$? How difficult is it to find $(m_1, m_2, m_3, \ldots, m_k)$ such that $f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus \cdots \oplus f_k(m_k) = 0$?

Motivation. These "generalized birthday problems" arise in a surprisingly wide variety of cryptanalytic tasks; see [9, Section 4]. The difficulty of solving these problems has a heavy influence on the choice of parameters for fast incremental hash functions, code-based public-key systems, etc. See my paper [3] for one example.

Previous answers. The standard answer for $k = 2$ is that finding a collision between $f_1(m_1) \bmod 2^B$ and $-f_2(m_2) \bmod 2^B$ takes time $2^{B/2}$.

The standard answer for $k > 2$ is that Wagner's "generalized birthday attack" in [9] takes time $2^{B/i}$ if $k = 2^{i-1}$; e.g., time $2^{B/5}$ if $k = 16$, improving on the time $2^{B/2}$ taken by traditional collision search. But this answer has several glaring deficiencies:

- 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.)

[*] Permanent ID of this document: 7cf298bebf853705133a84bea84d4a07. Date of this document: 2007.09.04. This work was carried out while the author was visiting Technische Universiteit Eindhoven.

- The standard answer fails to account for limits on machine cost. Wagner's algorithm needs a terrifyingly large machine with $2^{B/i}$ blocks of memory—for example, 2^{128} blocks if $B = 512$ and $k = 8$. For comparision, traditional collision search can be carried out by a tiny circuit, only slightly larger than a circuit to compute the functions $f_1, \ldots, f_k$. 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 $2^{B/2i}$, so the $2^{B/i}$ serial storage accesses in Wagner's algorithm take time $2^{3B/2i}$.

The bottom line is that Wagner's algorithm has a price-performance ratio on the scale of $2^{5B/2i}$: e.g., $2^{B/2}$ if $k = 16$. Wagner's algorithm is advertised as being faster than traditional methods for all $k \geq 4$; however, for (e.g.) $k = 8$, an attacker with any particular hardware budget and time budget has a much better chance of finding the desired $(m_1, m_2, \ldots, m_8)$ with traditional methods than with Wagner's algorithm.

Wagner in [9, 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 generalized-birthday attacks. In particular, this paper improves Wagner's algorithm, drastically reducing the time to $2^{B/(2i+1)}$ while reducing the machine size from $2^{B/i}$ to $2^{2B/(2i+1)}$.

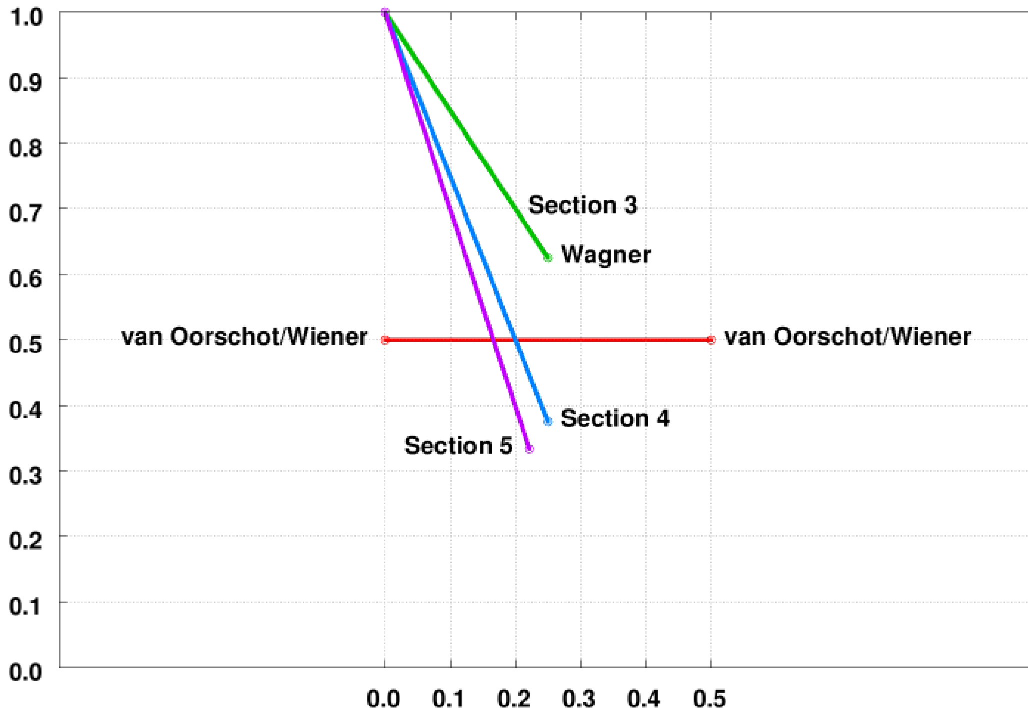
More generally, for each c between 0 and $2B/(2i+1)$, this paper presents an attack taking time 2^{B-ic} on a circuit of size 2^c. The price-performance ratio of the circuit is $2^{B-(i-1)c}$, rising from $2^{3B/(2i+1)}$ for $c = 2B/(2i+1)$ to $2^{B/2}$ for $c = B/(2i-2)$ and then higher as c drops. For $c < B/(2i-2)$ it is better to find collisions between $f_1(m_1) + \cdots + f_{k/2}(m_{k/2}) \bmod 2^B$ and $-f_{k/2+1}(m_{k/2+1}) - \cdots - f_k(m_k) \bmod 2^B$ using the parallel-collision-search circuit of van Oorschot and Wiener in [7], taking time approximately $2^{B/2-c}$ on a circuit of size 2^c.

Even more generally, for each c between 0 and $2B/(2i+1)$ and for each t between $c/2$ and $B-ic$, this paper presents an attack taking time 2^t on a circuit of size 2^c and having generic success probability approximately 2^{t+ic-B}. For $t > (i-2)c$ 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 $2^{2t+2c-B}$.

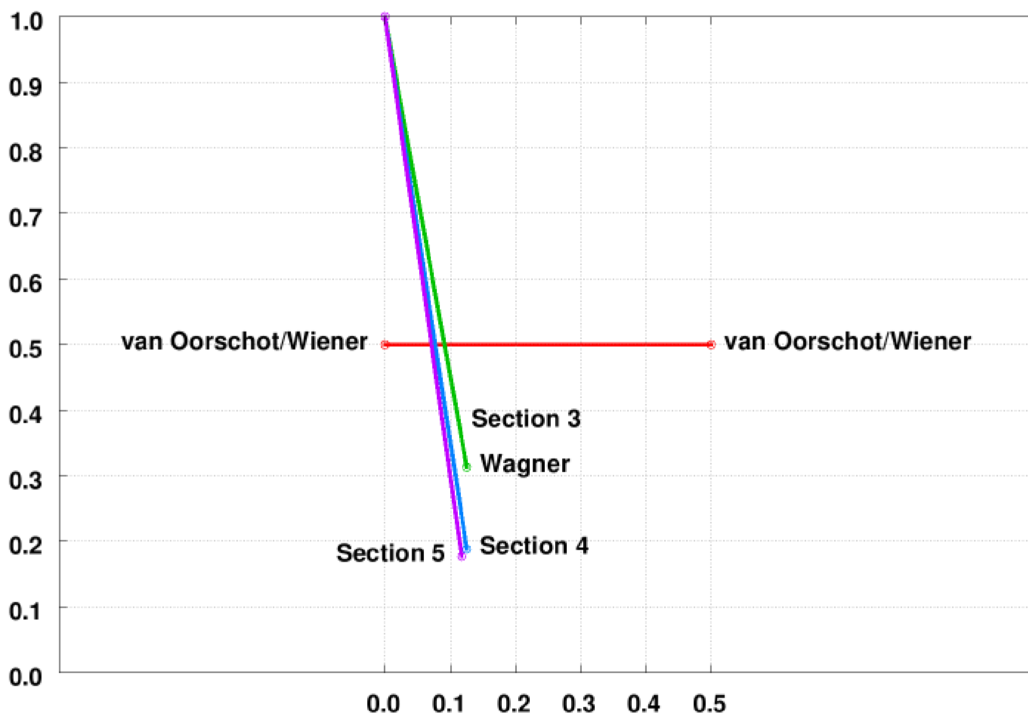
Section 2 reviews Wagner's attack. Sections 3, 4, and 5 present the new attack as the result of three improvements upon Wagner's attack.

I presented a preliminary version of this analysis, for the special case of 4-xor collisions, as part of my paper [3] at the ECRYPT Hash Workshop 2007. This paper supersedes [3, Section 2]. Exception: [3, Section 2] included an evaluation and improvement of constant factors for 4-xor collisions; this paper does not consider constant factors.

Graphs. The following graph shows how asymptotic price-performance ratio for $k = 8$ ($i = 4$) is affected by machine size and by choice of algorithm. The horizontal axis is c/B where the machine size is 2^c. The vertical axis is $(t+c)/B$ where the computation time is 2^t. The "Section 3" and "Section 4" and "Section 5" lines are for the algorithms presented in this paper.



The following graph is for $k = 128$ ($i = 8$), with the same axes.



Open questions. The new success probability 2^{t+ic-B} increases linearly with the time spent but as the ith power of the circuit size (up to size $2^{2B/(2i+1)}$). For comparison, the van Oorschot-Wiener success probability $2^{2t+2c-B}$ increases quadratically with time and quadratically with circuit size. What other functions of t, c can be achieved? Can one achieve $2t+(i-1)c-B$, for example, or $2t+ic-B$, or $it + c - B$?

The only obvious limit is $2^{kt+kc-B}$. A circuit of size 2^c cannot generate more than 2^{t+c} values $f_j(m_j)$ in time 2^t; there exist at most 2^{kt+kc} combinations $f_1(m_1) + f_2(m_2) + \cdots + f_k(m_k) \bmod 2^B$; there is a 0 among these combinations with probability at most $2^{kt+kc-B}$. Is there any better upper bound? The Brent-Kung theorem [4, Theorem 3.1] (predating [10, 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 sorting be avoided in inversion algorithms for the function $(m_1, \ldots, m_k) \mapsto f_1(m_1) + \cdots + f_k(m_k) \bmod 2^B$?

This paper focuses on asymptotic scalability. Choosing concrete parameter sizes requires a more detailed analysis for each application, taking account of the exact costs of circuits for sorting, evaluating f_j, et al.

Notes on price and performance. Price-performance ratio is the standard way to account for both machine cost and computation time. One divides the price—the machine cost, measured for example in Euros—by the performance—the computation speed, measured in computations per second. In other words, one multiplies the machine cost by the time for a computation.

The circuit-design literature often abbreviates price-performance ratio as "AT"; here A is circuit area and T is computation time. A few papers in the cryptographic literature use the strange name "full cost," achieving neither the clarity of "price-performance ratio" nor the brevity of "AT." Terminology aside, what makes the product AT useful is the simple fact that users who can afford area A' and time T' can carry out $A'T'/AT$ separate computations, assuming that A' is a multiple of A and T' is a multiple of T. Users generally receive the most benefit from machines built to minimize AT.

Limitations on machine cost sometimes force suboptimal price-performance ratios. To make this impact clear, this paper reports not merely the best price-performance ratio but also the range of options available for machine cost and computation time. One does not need to multiply A by T to understand the (A, T) improvements described in this paper.

There are other ways to measure the difficulty of computations. Algorithms in the literature are most commonly optimized for "operation count," where an "operation" can be as easy as adding two 64-bit integers or as difficult as randomly accessing a 2^{64}-byte array. The resulting algorithms, such as Wagner's algorithm, often achieve stunningly bad price-performance ratios compared to better-designed algorithms. There is nothing new about this basic observation—see, for example, the classic paper [4] on integer-multiplication circuits, or my paper [2] on integer-factorization circuits—but a wide variety of algorithms in the literature still need to be redesigned to optimize price-performance ratio.

2 Review of Wagner's algorithm

This section reviews Wagner's algorithm to find $m_1, \ldots, m_k$ with $f_1(m_1) + f_2(m_2) + \cdots + f_k(m_k) \bmod 2^B = 0$, when $k = 2^{i-1}$.

Choose $2^{B/i}$ different values of m_1 and $2^{B/i}$ different values of m_2. There are $2^{2B/i}$ pairs (m_1, m_2), and on average (generically) there are $2^{B/i}$ pairs such that $f_1(m_1) + f_2(m_2) \bmod 2^{B/i} = 0$. Find those pairs as follows: compute the $2^{B/i}$ values $(f_1(m_1) \bmod 2^{B/i}, m_1)$ and sort them into lexicographic order; compute the $2^{B/i}$ values $(-f_2(m_2) \bmod 2^{B/i}, m_2)$ and sort them into lexicographic order; merge the sorted lists to find all pairs (m_1, m_2) for which $f_1(m_1) \bmod 2^{B/i}$ matches $-f_2(m_2) \bmod 2^{B/i}$.

Wagner states that the sorting takes "$O(n \log n)$ time" where $n = 2^{B/i}$. 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/i)-bit strings actually takes time proportional to B/i, not constant time, so heap sort uses $O(n(\log n)^2)$ bit comparisons, not merely the claimed $O(n \log n)$; one can, on the other hand, replace heap sort with radix sort, eliminating a $\log n$ factor. 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 in Section 4.

After finding $2^{B/i}$ vectors (m_1, m_2) for which $f_1(m_1)+f_2(m_2) \bmod 2^{B/i} = 0$, use the same idea to find $2^{B/i}$ vectors (m_3, m_4) for which $f_3(m_3) + f_4(m_4) \bmod 2^{B/i} = 0$. Compute $f_1(m_1)+f_2(m_2) \bmod 2^{2B/i}$ for each (m_1, m_2), and $-f_3(m_3)-f_4(m_4) \bmod 2^{2B/i}$ for each (m_3, m_4); sort and merge to find, on average, $2^{B/i}$ vectors (m_1, m_2, m_3, m_4) for which $f_1(m_1) + f_2(m_2) + f_3(m_3) + f_4(m_4) \bmod 2^{2B/i} = 0$.

Use the same idea for $i-1$ levels of recursion to find, on average, $2^{B/i}$ vectors $(m_1, m_2, \ldots, m_k)$ for which $f_1(m_1) + f_2(m_2) + \cdots + f_k(m_k) \bmod 2^{(i-1)B/i} = 0$. Finally, compute $f_1(m_1) + f_2(m_2) + \cdots + f_k(m_k) \bmod 2^B$ for each vector $(m_1, m_2, \ldots, m_k)$. There will be, on average, 1 vector for which $f_1(m_1)+f_2(m_2)+\cdots + f_k(m_k) \bmod 2^B = 0$.

One can object that there is no reason for the chance of finding a vector to be as large as the average number of vectors found; but a more detailed analysis shows that the gap is negligible for large B/i. Limiting the intermediate lists to exactly $2^{B/i}$ vectors also makes very little difference in the success probability of the algorithm.

Generalizations. The above description assumed for simplicity that B is divisible by i. To handle the general case, replace B/i and $2B/i$ and so on by nearby integers.

One can find $m_1, \ldots, m_k$ with $f_1(m_1) \oplus f_2(m_2) \oplus \cdots \oplus f_k(m_k) = 0$ by essentially the same algorithm; see [9]. The differences between the algorithms are orthogonal to the speedups discussed in this paper.

3 Handling constraints on machine size

Wagner's algorithm needs $\Theta(2^{B/i}B/i)$ bits of memory to store $\Theta(2^{B/i})$ vectors, each having $\Theta(B/i)$ bits. What if the attacker cannot afford to pay for $\Theta(2^{B/i}B/i)$ bits of memory? This section adapts Wagner's algorithm to fit within a smaller machine.

Assume that the attacker can afford to store only 2^c vectors $(f_1(m_1) \bmod 2^{B/i}, m_1)$ rather than $2^{B/i}$ vectors, and only 2^c vectors $(-f_2(m_2) \bmod 2^{B/i}, m_2)$ rather than $2^{B/i}$ vectors. Here c is a parameter that will be reflected in the final machine cost.

The attacker now finds, on average, $2^{2c-B/i}$ vectors (m_1, m_2) for which $f_1(m_1) + f_2(m_2) \bmod 2^{B/i} = 0$. Together with $2^{2c-B/i}$ similar vectors (m_3, m_4) the attacker finds $2^{4c-3B/i}$ vectors (m_1, m_2, m_3, m_4) for which $f_1(m_1)+f_2(m_2)+f_3(m_3) + f_4(m_4) \bmod 2^{2B/i} = 0$. After another level of recursion the attacker finds $2^{8c-7B/i}$ vectors $(m_1, \ldots, m_8)$ for which $f_1(m_1)+\cdots+f_8(m_8) \bmod 2^{3B/i} = 0$. After $i-1$ levels the attacker finds $2^{kc-(k-1)B/i}$ vectors $(m_1, \ldots, m_k)$ for which $f_1(m_1) + \cdots + f_k(m_k) \bmod 2^{(i-1)B/i} = 0$, and therefore $2^{kc-kB/i}$ vectors $(m_1, \ldots, m_k)$ for which $f_1(m_1) + \cdots + f_k(m_k) \bmod 2^B = 0$.

Improvement: Increase the number of vectors (m_1, m_2) up to the machine capacity 2^c, by requiring only the bottom c bits (rather than B/i bits) of $f_1(m_1) + f_2(m_2)$ to be 0, Similarly increase the number of vectors (m_3, m_4). This increase by a factor of $2^{c-B/i}$ is reflected quadratically in the number of vectors found at the next level of recursion, quartically at the next level, etc., outweighing the loss of $c - B/i$ bits. Similarly increase the number of vectors (m_1, m_2, m_3, m_4) up to the machine capacity 2^c, and so on for each level of recursion.

The revised attack is as follows:

- Generate and sort 2^c vectors $(f_1(m_1) \bmod 2^c, m_1)$.
- Generate and sort 2^c vectors $(-f_2(m_2) \bmod 2^c, m_2)$.
- Merge to find 2^c vectors (m_1, m_2) for which $f_1(m_1) + f_2(m_2) \bmod 2^c = 0$.
- Similarly find 2^c vectors (m_3, m_4) for which $f_3(m_3) + f_4(m_4) \bmod 2^c = 0$.
- Sort and merge again to find 2^c vectors (m_1, m_2, m_3, m_4) for which $f_1(m_1)+f_2(m_2) + f_3(m_3) + f_4(m_4) \bmod 2^{2c} = 0$.
- Repeat for $i-1$ levels of recursion to find 2^c vectors $(m_1, m_2, \ldots, m_k)$ for which $f_1(m_1) + f_2(m_2) + \cdots + f_k(m_k) \bmod 2^{(i-1)c} = 0$, and therefore 2^{ic-B} vectors $(m_1, m_2, \ldots, m_k)$ for which $f_1(m_1)+f_2(m_2)+\cdots+f_k(m_k) \bmod 2^B = 0$.

One can also view the revised attack as follows: truncate each f_j to ic bits; use the original attack to find $(m_1, \ldots, m_k)$ for which $f_1(m_1) \oplus f_2(m_2) \oplus f_3(m_3) \oplus f_4(m_4) \bmod 2^{ic} = 0$; hope that this equation modulo 2^{ic} is actually an equation modulo 2^B.

4 Parallelization

This section improves the algorithm of Section 3 to take advantage of parallel processing.

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 [5] 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 [6], which uses a more complicated order of operations but reduces 8 to approximately 3.

In particular, one can sort 2^c vectors $(f_1(m_1) \bmod 2^c, m_1, 1)$ together with 2^c vectors $(-f_2(m_2) \bmod 2^c, m_2, 2)$ 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 and 2^c evaluations of f_2 and so on, but 2^c (or fewer) parallel processors can handle these evaluations at negligible cost for any reasonable choice of $f_1, f_2, \ldots, f_k$.

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^{ic-B} vectors $(m_1, m_2, \ldots, m_k)$ for which $f_1(m_1) + f_2(m_2) + \cdots + f_k(m_k) \bmod 2^B = 0$.

5 Precomputation

There is an imbalance in the parallelized algorithm of Section 4. This section corrects the imbalance, making the algorithm much more cost-effective.

Imbalance: 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 $f_1(m_1)$—typically one smaller than $2^{B-c/2}$. Similarly spend more time searching for useful values of m_2 etc.

This improvement increases the average number of solutions to the scale of $2^{c/2+ic-B}$. It increases the time for the attack to roughly $2^{c/2}$ compare-exchange

steps and roughly $2^{c/2}$ evaluations of f_j; still on the scale of $2^{c/2}$ overall, when evaluation of f_j 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 average number of solutions to the scale of 2^{t+ic-B}, as advertised in Section 1.

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 [6].
2. Daniel J. Bernstein, *Circuits for integer factorization: a proposal* (2001). URL: `http://cr.yp.to/papers.html#nfscircuit`. Citations in this document: §1.
3. Daniel J. Bernstein, *What output size resists collisions in a xor of independent expansions?*, ECRYPT Workshop on Hash Functions 2007 (2007). URL: `http://cr.yp.to/papers.html#expandxor`. Citations in this document: §1, §1, §1, §1.
4. 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, §1.
5. Manfred Schimmler, *Fast sorting on the instruction systolic array*, report 8709, Christian-Albrechts-Universität Kiel, 1987. Citations in this document: §4.
6. Claus P. Schnorr, Adi Shamir, *An optimal sorting algorithm for mesh-connected computers*, in [1] (1986), 255–261. Citations in this document: §4.
7. 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.
8. David Wagner, *A generalized birthday problem (extended abstract)*, in [11] (2002), 288–303; see also newer version [9]. URL: `http://www.cs.berkeley.edu/~daw/papers/genbday.html`.
9. David Wagner, *A generalized birthday problem (extended abstract) (long version)* (2002); see also older version [8]. URL: `http://www.cs.berkeley.edu/~daw/papers/genbday.html`. Citations in this document: §1, §1, §1, §2.
10. 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.
11. 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 [8].