FSBday: Implementing Wagner's generalized birthday attack against the SHA-3[⋆] round-1 candidate FSB

Daniel J. Bernstein[1], Tanja Lange[2], Ruben Niederhagen[3], Christiane Peters[2], and Peter Schwabe[2] [⋆⋆]

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to

[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
tanja@hyperelliptic.org, c.p.peters@tue.nl, peter@cryptojedi.org

[3] Lehrstuhl für Betriebssysteme, RWTH Aachen University
Kopernikusstr. 16, 52056 Aachen, Germany
ruben@polycephaly.org

Abstract. This paper applies generalized birthday attacks to the FSB compression function, and shows how to adapt the attacks so that they run in far less memory. In particular, this paper presents details of a parallel implementation attacking FSB_{48}, a scaled-down version of FSB proposed by the FSB submitters. The implementation runs on a cluster of 8 PCs, each with only 8GB of RAM and 700GB of disk. This situation is very interesting for estimating the security of systems against distributed attacks using contributed off-the-shelf PCs.

Keywords: SHA-3, Birthday, FSB – Wagner, not much Memory

1 Introduction

The hash function FSB [2] uses a compression function based on error-correcting codes. This paper describes, analyzes, and optimizes a parallelized generalized birthday attack against the FSB compression function.

This paper focuses on a reduced-size version FSB_{48} which was suggested as a training case by the designers of FSB. The attack against FSB_{48} has been implemented and carried out successfully, confirming our performance analysis. Our results allow us to accurately estimate how expensive a similar attack would be for full-size FSB.

A straightforward implementation of Wagner's generalized birthday attack [12] would need 20 TB of storage. However, we are running the attack on 8

[⋆] SHA-2 will soon retire, see [10]

[⋆⋆] This work was supported by the National Science Foundation under grant ITR-0716498, by the European Commission under Contract ICT-2007-216499 CACE, and by the European Commission under Contract ICT-2007-216646 ECRYPT II. Permanent ID of this document: ded1984108ff55330edb8631e7bc410c. Date: 2009.09.24.

nodes of the Coding and Cryptography Computer Cluster (CCCC) at Technische Universiteit Eindhoven, which has a total hard disk space of only 5.5 TB. We detail how we deal with this restricted background storage, by applying and generalizing ideas described by Bernstein in [6] and compressing partial results. We also explain the algorithmic measures we took to make the attack run as fast as possible, carefully balancing our code to use available RAM, network throughput, hard disk throughput and computing power.

We are to the best of our knowledge the first to give a detailed description of a full implementation of a generalized birthday attack. We plan to put all code described in this paper into the public domain to maximize reusability of our results.

Hash-function design. This paper achieves new speed records for generalized birthday attacks, and in particular for generalized birthday attacks against the FSB *compression* function. However, generalized birthday attacks are still much more expensive than generic attacks against the FSB *hash* function. "Generic attacks" are attacks that work against any hash function with the same output length.

The FSB designers chose the size of the FSB compression function so that a particular lower bound on the cost of generalized birthday attacks would be safely above the cost of generic attacks. Our results should not be taken as any indication of a security problem in FSB; the actual cost of generalized birthday attacks is very far above the lower bound stated by the FSB designers. It appears that the FSB compression function was designed too conservatively, with an unnecessarily large output length.

FSB was one of the 64 hash functions submitted to NIST's SHA-3 competition, and one of the 51 hash functions selected for the first round. However, FSB was significantly slower than most submissions, and was not one of the 14 hash functions selected for the second round. It would be interesting to explore smaller and thus faster FSB variants that remain secure against generalized birthday attacks.

Organization of the paper. In Section 2 we give a short introduction to Wagner's generalized birthday attack and Bernstein's adaptation of this attack to storage-restricted environments. Section 3 describes the FSB hash function to the extent necessary to understand our attack methodology. In Section 4 we describe our attack strategy which has to match the restricted hard disk space of our computer cluster. Section 5 details the measures we applied to make the attack run as efficiently as possible dealing with the bottlenecks mentioned before. We evaluate the overall cost of our attack in Section 6, and give cost estimates for a similar attack against full-size FSB in Section 7.

Naming conventions. Throughout the paper we will denote list j on level i as $L_{i,j}$. For both, levels and lists we start counting at zero.

Logarithms denoted as lg are logarithms to the base 2.

Additions of list elements or constants used in the algorithm are additions modulo 2.

In units such as GB, TB, PB and EB we will always assume base 1024 instead of 1000. In particular we give 700 GB as the size of a hard disk advertised as 750 GB.

2 Wagner's Generalized Birthday Attack

The generalized birthday problem, given 2^{i-1} lists containing B-bit strings, is to find 2^{i-1} elements — exactly one in each list — whose xor equals 0.

The special case $i = 2$ is the classic birthday problem: given two lists containing B-bit strings, find two elements — exactly one in each list — whose xor equals 0. In other words, find an element of the first list that equals an element of the second list.

This section describes a solution to the generalized birthday problem due to Wagner [12]. Wagner also considered generalizations to operations other than xor, and to the case of k lists when k is not a power of 2.

2.1 The tree algorithm

Wagner's algorithm builds a binary tree as described in this subsection starting from the input lists $L_{0,0}, L_{0,1}, \ldots, L_{0,2^{i-1}-1}$ (see Figure 4.1). The speed and success probability of the algorithm are analyzed under the assumption that each list contains $2^{B/i}$ elements chosen uniformly at random.

On level 0 take the first two lists $L_{0,0}$ and $L_{0,1}$ and compare their list elements on their least significant B/i bits. Given that each list contains about $2^{B/i}$ elements we can expect $2^{B/i}$ pairs of elements which are equal on those least significant B/i bits. We take the xor of both elements on all their B bits and put the xor into a new list $L_{1,0}$. Similarly compare the other lists — always two at a time — and look for elements matching on their least significant B/i bits which are xored and put into new lists. This process of *merging* yields 2^{i-2} lists containing each about $2^{B/i}$ elements which are zero on their least significant B/i bits. This completes level 0.

On level 1 take the first two lists $L_{1,0}$ and $L_{1,1}$ which are the results of merging the lists $L_{0,0}$ and $L_{0,1}$ as well as $L_{0,2}$ and $L_{0,3}$ from level 0. Compare the elements of $L_{1,0}$ and $L_{1,1}$ on their least significant $2B/i$ bits. As a result of the xoring in the previous level, the last B/i bits are already known to be 0, so it suffices to compare the next B/i bits. Since each list on level 1 contains about $2^{B/i}$ elements we again can expect about $2^{B/i}$ elements matching on B/i bits. We build the xor of each pair of matching elements and put it into a new list $L_{2,0}$. Similarly compare the remaining lists on level 1.

Continue in the same way until level $i - 2$. On each level j we consider the elements on their least significant $(j+1)B/i$ bits of which jB/i bits are known to be zero as a result of the previous merge. On level $i-2$ we get two lists containing about $2^{B/i}$ elements. The least significant $(i-2)B/i$ bits of each element in both lists are zero. Comparing the elements of both lists on their $2B/i$ remaining bits gives 1 expected match, i.e., one xor equal to zero. Since each element is the xor of elements from the previous steps this final xor is the xor of 2^{i-1} elements from the original lists and thus a solution to the generalized birthday problem.

2.2 Wagner in memory-restricted environments

A 2007 paper [6] by Bernstein includes two techniques to mount Wagner's attack on computers which do not have enough memory to hold all list entries. Various special cases of the same techniques also appear in a 2005 paper [4] by Augot, Finiasz, and Sendrier and in a 2009 paper [9] by Minder and Sinclair.

Clamping through precomputation. Suppose that there is space for lists of size only 2^b with $b < B/i$. Bernstein suggests to generate $2^{b\cdot(B-ib)}$ entries and only consider those of which the least significant $B - ib$ bits are zero.

We generalize this idea as follows: The least significant $B - ib$ bits can have an arbitrary value, this *clamping value* does not even have to be the same on all lists as long as the *sum* of all clamping values is zero. This will be important if an attack does not produce a collision. We then can simply restart the attack with different clamping values.

Clamping through precomputation may be limited by the maximal number of entries we can generate per list. Furthermore, halving the available storage space increases the precomputation time by a factor of 2^i.

Note that clamping some bits through precomputation might be a good idea even if enough memory is available as we can reduce the amount of data in later steps and thus make those steps more efficient.

After the precomputation step we apply Wagner's tree algorithm to lists containing bit strings of length B' where B' equals B minus the number of clamped bits. For performance evaluation we will only consider lists on level 0 *after* clamping through precomputation and then use B instead of B' for the number of bits in these entries.

Repeating the attack. Another way to mount Wagner's attack in memory-restricted environments is to carry out the whole computation with smaller lists leaving some bits at the end "uncontrolled". We then can deal with the lower success probability by repeatedly running the attack with different clamping values.

In the context of clamping through precomputation we can simply vary the clamping values used during precomputation. If for some reason we cannot clamp any bits through precomputation we can apply the same idea of changing clamping values in an arbitrary merge step of the tree algorithm. Note that any solution to the generalized birthday problem can be found by some choice of clamping values.

Expected number of runs. Wagner's algorithm, without clamping through precomputation, produces an expected number of exactly one collision. However this does not mean that running the algorithm necessarily produces a collision.

In general, the expected number of runs of Wagner's attack is a function of the number of remaining bits in the entries of the two input lists of the last merge step and the number of elements in these lists.

Assume that b bits are clamped on each level and that lists have length 2^b. Then the probability to have at least one collision after running the attack once is

$$P_{\text{success}} = 1 - \left(\frac{2^{B-(i-2)b} - 1}{2^{B-(i-2)b}}\right)^{2^{2b}},$$

and the expected number of runs $E(R)$ is

$$E(R) = \frac{1}{P_{\text{success}}}. \tag{2.1}$$

For larger values of $B - ib$ the expected number of runs is about 2^{B-ib}. We model the total time for the attack as being linear in the amount of data on level 0, i.e.,

$$t_W \in \Theta\left(2^{i-1}2^{B-ib}2^b\right). \tag{2.2}$$

Here 2^{i-1} is the number of lists, 2^{B-ib} is approximately the number of runs, and 2^b is the number of entries per list. Observe that this formula will usually underestimate the real time of the attack by assuming that all computations on subsequent levels are together still linear in the time required for computations on level 0.

Using Pollard iteration. If because of memory restrictions the number of uncontrolled bits is high, it may be more efficient to use a variant of Wagner's attack that uses Pollard iteration [8, Chapter 3, exercises 6 and 7].

Assume that $L_0 = L_1$, $L_2 = L_3$, etc., and that combinations $x_0 + x_1$ with $x_0 = x_1$ are excluded. The output of the generalized birthday attack will then be a collision between two distinct elements of $L_0 + L_2 + \cdots$.

We can instead start with only 2^{i-2} lists $L_0, L_2, \ldots$ and apply the usual Wagner tree algorithm, with a nonzero clamping constant to enforce the condition that $x_0 \neq x_1$. The number of clamped bits before the last merge step is now $(i-3)b$. The last merge step produces 2^{2b} possible values, the smallest of which has an expected number of $2b$ leading zeros, leaving $B - (i-1)b$ uncontrolled.

Think of this computation as a function mapping clamping constants to the final $B - (i-1)b$ uncontrolled bits and apply Pollard iteration to find a collision between the output of two such computations; combination then yields a collision of 2^{i-1} vectors.

As Pollard iteration has square-root running time, the expected number of runs for this variant is $2^{B/2-(i-1)b/2}$, each taking time $2^{i-2}2^b$ (cmp. (2.2)), so the expected running time is

$$t_{PW} \in \Theta\left(2^{i-2}2^{B/2-(i-1)b/2+b}\right). \tag{2.3}$$

The Pollard variant of the attack becomes more efficient than plain Wagner with repeated runs if $B > (i+2)b$.

3 The FSB Hash Function

In this section we briefly describe the construction of the FSB hash function. Since we are going to attack the function we omit details which are necessary for implementing the function but do not influence the attack. The second part of this section gives a rough description of how to apply Wagner's generalized birthday attack to find collisions of the compression function of FSB.

3.1 Details of the FSB hash function

The Fast Syndrome Based hash function (FSB) was introduced by Augot, Finiasz and Sendrier in 2003. See [3], [4], and [2]. The security of FSB's compression function relies on the difficulty of the "Syndrome Decoding Problem" from coding theory.

The FSB hash function processes a message in three steps: First the message is converted by a so-called domain extender into suitable inputs for the compression function which digests the inputs in the second step. In the third and final step the Whirlpool hash function designed by Barreto and Rijmen [5] is applied to the output of the compression function in order to produce the desired length of output.

Our goal in this paper is to investigate the security of the compression function. We do not describe the domain extender, the conversion of the message to inputs for the compression function, or the last step involving Whirlpool.

The compression function. The main parameters of the compression function are called n, r and w. We consider n strings of length r which are chosen uniformly at random and can be written as an $r \times n$ binary matrix H. Note that the matrix H can be seen as the parity check matrix of a binary linear code. The FSB proposal [2] actually specifies a particular structure of H for efficiency; we do not consider attacks exploiting this structure.

An n-bit string of weight w is called *regular* if there is exactly a single 1 in each interval $[(i-1)\frac{n}{w}, i\frac{n}{w}-1]_{1 \leq i \leq w}$. We will refer to such an interval as a *block*. The input to the compression function is a regular n-bit string of weight w.

The compression function works as follows. The matrix H is split into w blocks of n/w columns. Each non-zero entry of the input bit string indicates exactly one column in each block. The output of the compression function is an r-bit string which is produced by computing the xor of all the w columns of the matrix H indicated by the input string.

Preimages and collisions. A preimage of an output of length r of one round of the compression function is a regular n-bit string of weight w. A collision occurs if there are $2w$ columns of H — exactly two in each block — which add up to zero.

Finding preimages or collisions means solving two problems coming from coding theory: finding a preimage means solving the Regular Syndrome Decoding problem and finding collisions means solving the so-called 2-regular Null-Syndrome Decoding problem. Both problems were defined and proven to be NP-complete in [4].

Parameters. We follow the notation in [2] and write $\mathrm{FSB}_{\texttt{length}}$ for the version of FSB which produces a hash value of length `length`. Note that the output of the compression function has r bits where r is considerably larger than `length`.

NIST demands hash lengths of 160, 224, 256, 384, and 512 bits, respectively. Therefore the SHA-3 proposal contains five versions of FSB: FSB_{160}, FSB_{224}, FSB_{256}, FSB_{384}, and FSB_{512}. We list the parameters for those versions in Table 7.1.

The proposal also contains FSB_{48}, which is a reduced-size version of FSB and the main attack target in this paper. The binary matrix H for FSB_{48} has dimension $192 \times 3 \cdot 2^{17}$; i.e., r equals 192 and n is $3 \cdot 2^{17}$. In each round a message chunk is converted into a regular $3 \cdot 2^{17}$-bit string of Hamming weight $w = 24$. The matrix H contains 24 blocks of length 2^{14}. Each 1 in the regular bit string indicates exactly one column in a block of the matrix H. The output of the compression function is the xor of those 24 columns.

A pseudo-random matrix. In our attack against FSB_{48} we consider a pseudo-random matrix H which we constructed as described in [2, Section 1.2.2]: H consists of 2048 submatrices, each of dimension 192×192. For the first submatrix we consider a slightly larger matrix of dimension 197×192. Its first column consists of the first 197 digits of π where each digit is taken modulo 2. The remaining 191 columns of this submatrix are cyclic shifts of the first column. The matrix is then truncated to its first 192 rows which form the first submatrix of H. For the second submatrix we consider digits 198 up to 394 of π. Again we build a 197×192 bit matrix where the first column corresponds to the selected digits (each taken modulo 2) and the remaining columns the cyclic shifts of the first column. Truncating to the first 192 rows yields the second block matrix of H. The remaining submatrices are constructed in the same way.

We emphasize that this is one possible choice for the matrix H. The attack described in our paper does not make use of the structure of this particular matrix. We use this construction in our implementation since it is also contained in the FSB reference implementation submitted to NIST by the FSB designers.

3.2 Attacking the compression function of FSB_{48}

Coron and Joux pointed out in [7] that Wagner's generalized birthday attack can be used to find preimages and collisions in the compression function of FSB. The following paragraphs present a slightly streamlined version of the attack of [7] in the case of FSB_{48}.

Determining the number of lists for a Wagner attack on FSB_{48}. A collision for FSB_{48} is given by 48 columns of the matrix H which add up to zero; the collision has exactly two columns per block. Each block contains 2^{14} columns and each column is a 192-bit string.

We choose 16 lists to solve this particular 48-sum problem. Each list entry will be the xor of three columns coming from one and a half blocks. This ensures that we do not have any overlaps, i.e., more than two columns coming from one matrix block in the end. We assume that taking sums of the columns of H

does not bias the distribution of 192-bit strings. Applying Wagner's attack in a straightforward way means that we need to have at least $2^{\lceil 192/5 \rceil}$ entries per list. By clamping away 39 bits in each step we expect to get at least one collision after one run of the tree algorithm.

Building lists. We build 16 lists containing 192-bit strings each being the xor of three distinct columns of the matrix H. We select each triple of three columns from one and a half blocks of H in the following way:

List $L_{0,0}$ contains the sums of columns i_0, j_0, k_0, where columns i_0 and j_0 come from the first block of 2^{14} columns, and column k_0 is picked from the following block with the restriction that it is taken from the first half of it. Since we cannot have overlapping elements we get about 2^{27} sums of columns i_0 and j_0 coming from the first block. These two columns are then added to all possible columns k_0 coming from the first 2^{13} elements of the second block of the matrix H. In total we get about 2^{40} elements for $L_{0,0}$.

We note that by splitting every second block in half we neglect several solutions of the 48-xor problem. For example, a solution involving two columns from the first half of the second block cannot be found by this algorithm. We justify our choice by noting that fewer lists would nevertheless require more storage and a longer precomputation phase to build the lists.

The second list $L_{0,1}$ contains sums of columns i_1, j_1, k_1, where column i_1 is picked from the second half of the second block of H and j_1 and k_1 come from the third block of 2^{14} columns. This again yields about 2^{40} elements.

Similarly, we construct the lists $L_{0,2}$, $L_{0,3}, \ldots$, $L_{0,15}$.

For each list we generate more than twice the amount needed for a straightforward attack as explained above. In order to reduce the amount of data for the following steps we note that about $2^{40}/4$ elements are likely to be zero on their least significant two bits. Clamping those two bits away should thus yield a list of 2^{38} bit strings. Note that since we know the least significant two bits of the list elements we can ignore them and regard the list elements as 190-bit strings. Now we expect that a straightforward application of Wagner's attack to 16 lists with about $2^{190/5}$ elements yields a collision after completing the tree algorithm.

Note on complexity in the FSB proposal. The SHA-3 proposal estimates the complexity of Wagner's attack as described above as $2^{r/i}r$ where 2^{i-1} is the number of lists used in the algorithm. This does not take memory into account, and in general is an underestimate of the work required by Wagner's algorithm; i.e., attacks of this type against FSB are more difficult than claimed by the FSB designers.

Note on information-set decoding. The FSB designers say in [2] that Wagner's attack is the fastest known attack for finding preimages, and for finding collisions for small FSB parameters, but that another attack — information-set decoding — is better than Wagner's attack for finding collisions for large FSB parameters.

In general, information-set decoding can be used to find an n-bit string of weight 48 indicating 48 columns of H which add up to zero. Information-set decoding will not take into account that we look for a *regular* n-bit string. The only known way to obtain a regular n-bit string is running the algorithm repeatedly until the output happens to be regular. Thus, the running times given in [2] provide certainly lower bounds for information-set decoding, but in practice they are not likely to hold.

4 Attack Strategy

In this section we will discuss the necessary measures we took to mount the attack on our cluster. We will start with an evaluation of available and required storage.

4.1 How large is a list entry?

The number of bytes required to store one list entry depends on how we represent the entry. We considered four different ways of representing an entry:

Value-only representation. The obvious way of representing a list entry is as a 192-bit string, the xor of columns of the matrix. Bits we already know to be zero of course do not have to be stored, so on each level of the tree the number of bits per entry decreases by the number of bits clamped on the previous level. Ultimately we are not interested in the *value* of the entry — we know already that in a successful attack it will be all-zero at the end — but in the column positions in the matrix that lead to this all-zero value. However, we will show in Section 4.3 that computations only involving the *value* can be useful if the attack has to be run multiple times due to storage restrictions.

Value-and-positions representation. If enough storage is available we can store positions in the matrix alongside the value. Observe that unlike storage requirements for *values* the number of bytes for *positions* increases with increasing levels, and becomes dominant for higher levels.

Compressed positions. Instead of storing full positions we can save storage by only storing, e.g., positions modulo 256. After the attack has successfully finished the full position information can be computed by checking which of the possible positions lead to the appropriate intermediate results on each level.

Dynamic recomputation. If we keep full positions we do not have to store the value at all. Every time we need the value (or parts of it) it can be dynamically recomputed from the positions. In each level the size of a single entry doubles (because the number of positions doubles), the expected number of entries per list remains the same but the number of lists halves, so the total amount of data is the same on each level when using dynamic recomputation. As discussed in Section 3 we have 2^{40} possibilities to choose columns to produce entries of a list, so we can encode the positions on level 0 in 40 bits (5 bytes).

Observe that we can switch between representations during computation if at some level another representation becomes more efficient: We can switch between value-and-position representation to compressed-positions representation and back. We can switch from one of the above to compressed positions and we can switch from any other representation to value-only representation.

4.2 What list size can we handle?

To estimate the storage requirements it is convenient to consider *dynamic recomputation* (storing positions only) because in this case the amount of required storage is constant over all levels and this representation has the smallest memory consumption on level 0.

As described in Section 3.2 we can start with 16 lists of size 2^{38}, each containing bit strings of length $r' = 190$. However, storing 16 lists with 2^{38} entries, each entry encoded in 5 bytes requires 20 TB of storage space.

The computer cluster used for the attack consists of 8 nodes with a storage space of 700 GB each. Hence, we have to adapt our attack to cope with total storage limited to 5.5 TB.

On the first level we have 16 lists and as we need at least 5 bytes per list entry we can handle at most $5.5 \cdot 2^{40}/2^{4}/5 = 1.1 \times 2^{36}$ entries per list. Some of the disk space is used for the operating system and so a straightforward implementation would use lists of size 2^{36}. First computing one half tree and switching to compressed-positions representation on level 2 would still not allow us to use lists of size 2^{37}.

We can generate at most 2^{40} entries per list so following [6] we could clamp 4 bits during list generation, giving us 2^{36} values for each of the 16 lists. These values have a length of 188 bits represented through 5 bytes holding the positions from the matrix. Clamping 36 bits in each of the 3 steps leaves two lists of length 2^{36} with 80 non-zero bits. According to (2.1) we thus expect to run the attack 256.5 times until we find a collision.

The only way of increasing the list size to 2^{37} and thus reduce the number of runs is to use value-only representation on higher levels.

4.3 The strategy

The main idea of our attack strategy is to distinguish between the task of finding clamping constants that yield a final collision and the task of actually computing the collision.

Finding appropriate clamping constants. This task does not require storing the positions, since we only need to know whether we find a collision with a particular set of clamping constants; we do not need to know which matrix positions give this collision.

Whenever storing the value needs less space we can thus *compress* entries by switching representation from positions to values. As a side effect this speeds up the computations because less data has to be loaded and stored.

Starting from lists $L_{0,0}, \ldots, L_{0,7}$, each containing 2^{37} entries we first compute list $L_{3,0}$ (see Figure 4.1) on 8 nodes. This list has entries with 78 remaining bits each. As we will describe in Section 5, these entries are presorted on hard disk according to 9 bits that do not have to be stored. Another 3 bits are determined by the node holding the data (see also Section 5) so only 66 bits or 9 bytes of each entry have to be stored, yielding a total storage requirement of 1152 GB versus 5120 GB necessary for storing entries in positions-only representation.

We then continue with the computation of list $L_{2,2}$, which has entries of 115 remaining bits. Again 9 of these bits do not have to be stored due to presorting, 3 are determined by the node, so only 103 bits or 13 bytes have to be stored, yielding a storage requirement of 1664 GB instead of 2560 GB for uncompressed entries.

After these lists have been stored persistently on disk, we proceed with the computation of list $L_{2,3}$, then $L_{3,1}$ and finally check whether $L_{4,0}$ contains at least one element. These computations require another 2560 GB.

Therefore total amount of storage sums up to 1152 GB + 1664 GB + 2560 GB = 5376 GB; obviously all data fits onto the hard disk of the 8 nodes.

If a computation with given clamping constants is not successful, we change clamping constants only for the computation of $L_{2,3}$. The lists $L_{3,0}$ and $L_{2,2}$ do not have to be computed again. All combinations of clamping values for lists $L_{0,12}$ to $L_{0,15}$ summing up to 0 are allowed. Therefore there is a large amount of valid clamp bit combinations.

With 37 bits clamped on every level and 3 clamped through precomputation we are left with 4 uncontrolled bits and therefore, according to (2.1), expect 16.5 runs of this algorithm.

Computing the matrix positions of the collision. In case of success we know which clamping constants we can use and we know which value in the lists $L_{3,0}$ and $L_{3,1}$ yields a final collision. Now we can recompute lists $L_{3,0}$ and $L_{3,1}$ without compression to obtain the positions. For this task we decided to store only positions and use dynamic recomputation. On level 0 and level 1 this is the most space-efficient approach and we do not expect a significant speedup from switching to compressed-positions representation on higher levels. In total one half-tree computation requires 5120 GB of storage, hence, they have to be performed one after the other on 8 nodes.

The (re-)computation of lists $L_{3,0}$ and $L_{3,2}$ is an additional time overhead over doing all computation on list positions in the first place. However, this cost is incurred only once, and is amply compensated for by the reduced data volume in previous steps. See Section 5.2.

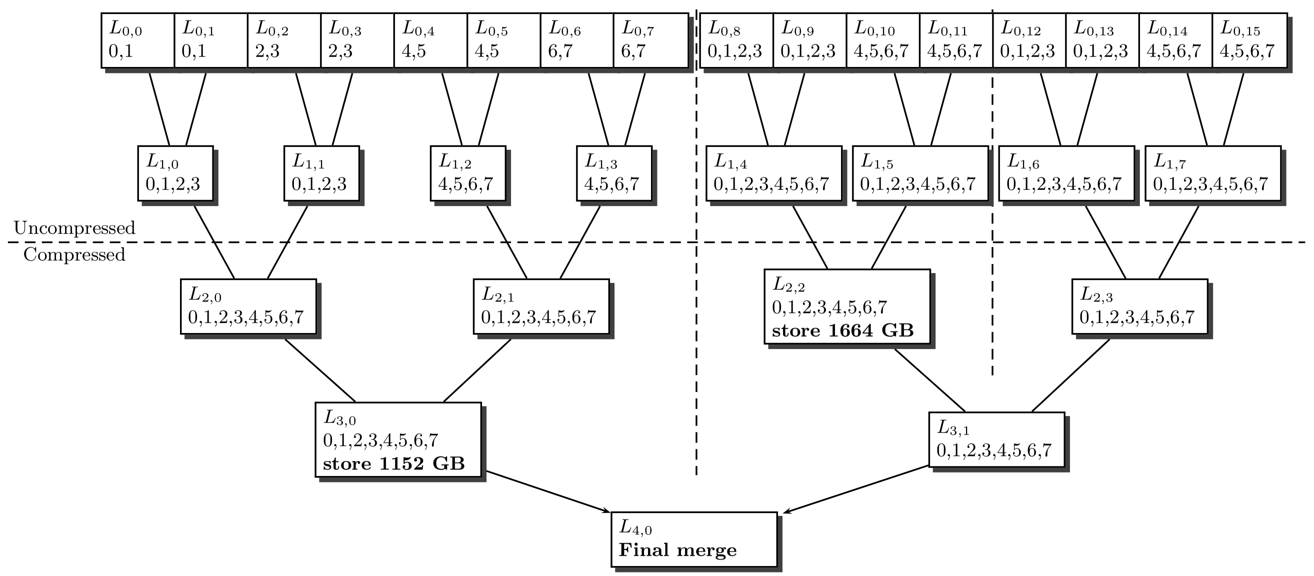

Fig. 4.1. Structure of the attack: in each box the upper line denotes the list, the lower line gives the nodes holding fractions of this list

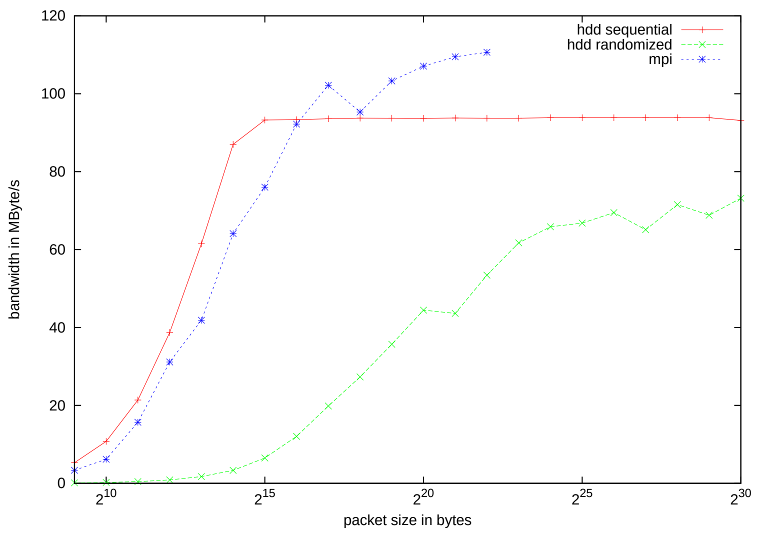

Fig. 5.1. Micro-benchmarks measuring hard disk and network throughput.

5 Implementing the Attack

The computation platform for this particular implementation of Wagner's generalized birthday attack on FSB is an eight-node cluster of conventional desktop PCs. Each node has an Intel Core 2 Quad Q6600 CPU with a clock rate of 2.40 GHz and direct fully cached access to 8 GB of RAM. About 700 GB mass storage are provided by a Western Digital SATA hard disk with 20 GB reserved for system and user data. The nodes are connected via switched Gigabit Ethernet using Marvell PCI-E adapter cards.

We chose MPI as communication model for the implementation. This choice has several virtues:

– MPI provides an easy interface to start the application on all nodes and to initialize the communication paths.
– MPI offers synchronous message-based communication primitives.
– MPI is a broadly accepted standard for HPC applications and is provided on a multitude of different platforms.

We decided to use MPICH2 [1] which is an implementation of the MPI 2.0 standard from the University of Chicago. MPICH2 provides an Ethernet-based back end for the communication with remote nodes and a fast shared-memory-based back end for local data exchange.

We implemented two micro-benchmarks to measure hard disk and network throughput. The results of these benchmarks are shown in Figure 5.1. Note that we measure hard disk throughput directly on the device, circumventing the filesystem, to reach peak performance of the hard disk. We measured both sequential and randomized access to the disk.

The rest of this section explains how we parallelized and streamlined Wagner's attack to make the best of the available hardware.

5.1 Parallelization

Most of the time in the attack is spent on determining the right clamping constants. As described in Section 4 this involves computations of several partial trees, e.g., the computation of $L_{3,0}$ from lists $L_{0,0}, \dots, L_{0,7}$ (half tree) or the computation of $L_{2,2}$ from lists $L_{0,8}, \dots, L_{0,11}$ (quarter tree). Other computations do not start with lists of level 0, the computation of list $L_{3,1}$ for example is computed from the (previously computed and stored) lists $L_{2,2}$ and $L_{2,3}$.

Lists of level 0 are generated with the current clamping constants. On every level, each list is sorted and afterwards merged with its neighboring list giving the entries for the next level. The sorting and merging is repeated until the final list of the partial tree is computed.

Distributing data over nodes. This algorithm is parallelized by distributing fractions of lists over the nodes in a way that each node can perform sort and merge locally on two lists. On each level of the computation, each node contains fractions of two lists. The lists on level j are split between n nodes according to $\lg(n)$ bits of each value. For example when computing the left half-tree, on level 0, node 0 contains all entries of lists 0 and 1 ending with a zero bit (in the bits not controlled by initial clamping), and node 1 contains all entries of lists 0 and 1 ending with a one bit.

Therefore, from the view of one node, on each level the fractions of both lists are loaded from hard disk, the entries are sorted and the two lists are merged. The newly generated list is split into its fractions and these fractions are sent over the network to their associated nodes. There the data is received and stored onto the hard disk. The continuous dataflow of this implementation is depicted in Figure 5.2.

Presorting into parts. To be able to perform the sort in memory, incoming data is presorted into one of 512 parts according to the 9 least significant bits of the current sort range. This leads to an expected part size for uncompressed entries of 640 MB (0.625 GB) which can be loaded into main memory at once to be sorted further. The benefit of presorting the entries before storing them is:

1. We can sort a whole fraction, that exceeds the size of the memory, by sorting its presorted parts independently.
2. Two adjacent parts of the two lists on one node (with the same presort-bits) can be merged directly after they are sorted.
3. We can save 9 bits when compressing entries to value-only representation.

Merge. The merge is implemented straightforwardly. If blocks of entries in both lists share the same value then all possible combinations are generated: specifi-

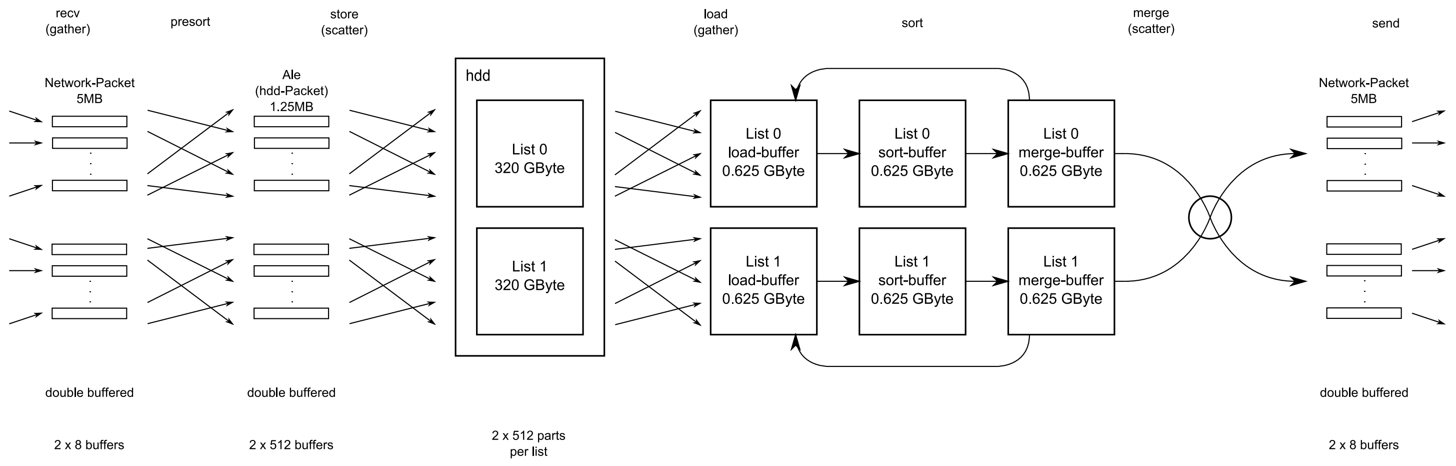

Fig. 5.2. Data flow during the computation within one half-tree

cally, if a b-bit string appears in the compared positions in c_1 entries in the first list and c_2 entries in the second list then all $c_1 c_2$ xors appear in the output list.

5.2 Efficient implementation

Cluster computation imposes three main bottlenecks:

- the computational power and memory latency of the CPUs for computation-intensive applications
- limitations of network throughput and latency for communication-intensive applications
- hard disk throughput and latency for data-intensive applications

Wagner's algorithm imposes hard load on all of these components: a large amount of data needs to be sorted, merged and distributed over the nodes occupying as much storage as possible. Therefore, demand for optimization is primarily determined by the slowest component in terms of data throughput; latency generally can be hidden by pipelining and data prefetch.

Finding bottlenecks. Our benchmarks show that, for sufficiently large packets, the performance of the system is mainly bottlenecked by hard disk throughput (cmp. Figure 5.1). Since the throughput of MPI over Gigabit Ethernet is higher than the hard disk throughput for packet sizes larger than 2^{16} bytes and since the same amount of data has to be sent that needs to be stored, no performance penalty is expected by the network for this size of packets.

Therefore, our first implementation goal was to design an interface to the hard disk that permits maximum hard disk throughput. The second goal was to optimize the implementation of sort and merge algorithms up to a level where the hard disks are kept busy at peak throughput.

Persistent data storage. Since we do not need any caching-, journaling- or even filing-capabilities of conventional filesystems, we implemented a throughput-optimized filesystem, which we call *AleSystem*. It provides fast and direct access to the hard disk and stores data in portions of *Ales*. Each cluster node has one large unformatted data partition `sda1`, which is directly opened by the AleSystem using native Linux file I/O. Caching is deactivated by using the open flag O_DIRECT: after data has been written, it is not read for a long time and does not benefit from caching. All administrative information is persistently stored as a file in the native Linux filesystem an mapped into the virtual address space of the process. On sequential access, the throughput of the AleSystem reaches about 90 MB/s which is roughly the maximum that the hard disk permits.

Tasks and threads. Since our cluster nodes are driven by quad-core CPUs, the speed of the computation is primarily based on multi-threaded parallelization. On the one side, the receive-/presort-/store, on the other side, the load-/sort-/merge-/send-tasks are pipelined. At the current state of the implementation, we have several threads for sending/receiving data and for running the AleSystem. The core of the implementation is given by five threads which process the main computation. There are two threads which have the task to presort incoming data (one thread for each list). Furthermore, sorting is parallelized with two threads (one thread for each list) and for the merge task we have one more thread.

Memory layout. Given this task distribution, the size of necessary buffers can be defined. The micro-benchmarks show that bigger buffers generally lead to higher throughput. However, the sum of all buffer sizes is limited by the size of the available RAM. For the list parts we need 6 buffers; we need two times 2×8 network buffers for double-buffered send and receive, which results in 32 network buffers. To presort the entries double-buffered into 512 parts of two lists, we need 2048 ales.

When a part is loaded from disk, its ales are treated as a continuous field of entries. Therefore, each ale must be completely filled with entries; no data padding at the end of each ale is allowed. Thus, we must pick a size for the ales which enables the ales to be completely filled independent of the varying size of entries over the whole run of the program. Valid sizes of entries are 5, 10, 20, and 40 bytes when storing positions and 5, 10, 13, and 9 bytes when storing compressed entries. Furthermore, since we access the hard disk using DMA, the size of each ale must be a multiple of 512 bytes. A multiple of a full memory page (4096 bytes) is not mandatory.

For these reasons, the size of one ale must be a multiple of $5 \times 9 \times 13 \times 512$. The size of network packets does not necessarily need to be a multiple of all possible entry sizes; if network packets happen not to be completely filled we merely waste some bytes of bandwidth.

In the worst case, on level 0 one list containing 2^{37} entries is distributed over 2 nodes and presorted into 512 parts; thus the size of each part should be larger than $2^{37}/2/512 \times 5$ bytes $= 640$ MB. The actual size of each part depends on the size of the ales since it must be an integer multiple of the ale size.

Finally, we chose a size of $2^{20} \cdot 5$ bytes $= 5$ MB for the network packets summing up to 160 MB, a size of $5 \times 9 \times 13 \times 512 \times 5 = 1497600$ bytes (about 1.4 MB) for the ales giving a memory demand of 2.9 GB for 2048 ales, and a size of $5 \times 9 \times 13 \times 512 \times 5 \times 512 = 766771200$ bytes (731.25 MB) for the parts summing up to 4.3 GB for 6 parts. Overall our implementation requires about 7.4 GB of RAM leaving enough space for the operating system and additional data as stack and the administrative data for the AleSystem.

Efficiency and further optimizations. Using our rough splitting of tasks to threads, we reach an average CPU usage of about 60% up to 80% peak. At the current optimization state, our average hard disk throughput is about 40 MB/s. The hard disk micro-benchmark (see Figure 5.1) shows that an average throughput between 45 MB/s and 50 MB/s should be feasible for packet sizes of 1.25 MB. Since sorting is the most complex task, we plan to further parallelize sorting to be able to use 100% of the CPU if the hard disk permits higher data transfer. We expect that further parallelization of the sort task will increase CPU data throughput on sort up to about 50 MB/s. That should suffice for maximum hard disk throughput.

6 Results

We have successfully carried out our FSB_{48} attack. This section presents (1) our estimates, before starting the attack, of the amount of time that the attack would need; (2) measurements of the amount of time actually consumed by the attack; and (3) comments on how different amounts of storage would have changed the attack time.

6.1 Cost estimates

Step one. As described before the first major step is to compute a set of clamping values which leads to a collision. In this first step entries are stored by positions on level 0 and 1 and from level 2 on list entries consist of values.

Computation of list $L_{3,0}$ takes about 32h and list $L_{2,2}$ about 14h, summing up to 46h. These computations need to be done only once.

The time needed to compute list $L_{2,3}$ is about the same as for $L_{2,2}$ (14h), list $L_{3,1}$ takes about 4h and checking for a collision in lists $L_{3,0}$ and $L_{3,1}$ on level 4 about another 3.5h, summing up to about 21.5h. The expected value of repetitions of these steps is 16.5 and and we thus expected them to take about 355h.

Step two. Finally, computing the matrix positions after finding a collision, requires recomputation with uncompressed lists. We only have to compute the entries of lists $L_{3,0}$ and $L_{3,1}$ until we have found the entry that yields the collision. In the worst case this computation with uncompressed (positions-only) entries takes 33h for each half-tree, summing up to 66h.

Total. Overall we expected to find a collision for the FSB_{48} compression function using our algorithm and cluster in 467h or about 19.5 days.

6.2 Cost measurements

We ran the code described above on our cluster and were lucky: In step one we found clamping constants after only five iterations (instead of the expected 16.5). In total the first phase of the attack took 5 days, 13 hours and 20 minutes.

Recomputation of the positions in $L_{3,0}$ took 1 day, 8 hours and 22 minutes and recomputation of the positions in $L_{3,1}$ took 1 day, 2 hours and 11 minutes. In total the attack took 7 days, 23 hours and 53 minutes.

Recall that the matrix used in the attack is the pseudo-random matrix defined in Section 3. We found that matrix positions (734, 15006, 20748, 25431, 33115, 46670, 50235, 51099, 70220, 76606, 89523, 90851, 99649, 113400, 118568, 126202, 144768, 146047, 153819, 163606, 168187, 173996, 185420, 191473 198284, 207458, 214106, 223080, 241047, 245456, 247218, 261928, 264386, 273345, 285069, 294658, 304245, 305792, 318044, 327120, 331742, 342519, 344652, 356623, 364676, 368702, 376923, 390678) yield a collision.

6.3 Time-storage tradeoffs

As described in Section 4, the main restriction on the attack strategy was the total amount of background storage.

If we had 10496 GB of storage at hand we could have handled lists of size 2^{38}, again using the compression techniques described in Section 4. As described in Section 4 this would give exactly one expected collision in the last merge step and thus reduce the expected number of required runs to find the right clamping constants from 16.5 to 1.58. With a total storage of 20 TB we could have run a straightforward Wagner attack without compression which would eliminate the need to recompute two half trees at the end.

Increasing the size of the background storage even further would eventually allow to store list entry values alongside the positions and thus eliminate the need for dynamic recomputation. However, the performance of the attack is bottlenecked by hard disk throughput rather than CPU time so we don't expect any improvement through this measure.

On clusters with even less background storage the computation time will (asymptotically) increase by a factor of 16 with each halving of the storage size. For example a cluster with 2688 GB of storage can only handle lists of size 2^{36}. The attack would then require (expected) 256.5 computations to find appropriate clamping constants.

Of course the time required for one half-tree computation depends on the amount of data. As long as the performance is mainly bottlenecked by hard-disk (or network) throughput the running time is linearly dependent on the amount of data, i.e., a Wagner computation involving 2 half-tree computations with lists of size 2^{38} is about 4.5 times as fast as a Wagner computation involving 18 half-tree computations with lists of size 2^{37}.

7 Scalability Analysis

The attack described in this paper including the variants discussed in Section 6 are much more expensive in terms of time and especially memory than a brute-force attack against the 48-bit hash function FSB_{48}.

This section gives estimates of the power of Wagner's attack against the larger versions of FSB, demonstrating that the FSB design overestimated the power of the attack. Table 7.1 gives the parameters of all FSB hash functions.

A straightforward Wagner attack against FSB_{160} uses 16 lists of size 2^{127} containing elements with 632 bits. The entries of these lists are generated as xors of 10 columns from 5 blocks, yielding 2^{135} possibilities to generate the entries. Precomputation includes clamping of 8 bits. Each entry then requires 135 bits of storage so each list occupies more than 2^{131} bytes. For comparison, the largest currently available storage systems offer a few petabytes (2^{50} bytes) of storage.

To limit the amount of memory we can instead generate, e.g., 32 lists of size 2^{60}, where each list entry is the xor of 5 columns from 2.5 blocks, with 7 bits clamped during precomputation. Each list entry then requires 67 bits of storage.

	n	w	r	Number of lists	Size of lists	Bits per entry	Total storage	Time
FSB$_{48}$	3×2^{17}	24	192	16	2^{38}	190	$5 \cdot 2^{42}$	$5 \cdot 2^{42}$
FSB$_{160}$	7×2^{18}	112	896	16	2^{127}	632	$17 \cdot 2^{131}$	$17 \cdot 2^{131}$
				16 (Pollard)	2^{60}	630	$9 \cdot 2^{64}$	$9 \cdot 2^{224}$
FSB$_{224}$	2^{21}	128	1024	16	2^{177}	884	$24 \cdot 2^{181}$	$24 \cdot 2^{181}$
				16 (Pollard)	2^{60}	858	$13 \cdot 2^{64}$	$13 \cdot 2^{343}$
FSB$_{256}$	23×2^{16}	184	1472	16	2^{202}	1010	$27 \cdot 2^{206}$	$27 \cdot 2^{206}$
				16 (Pollard)	2^{60}	972	$14 \cdot 2^{64}$	$14 \cdot 2^{386}$
				32 (Pollard)	2^{56}	1024	$18 \cdot 2^{60}$	$18 \cdot 2^{405}$
FSB$_{384}$	23×2^{16}	184	1472	16	2^{291}	1453	$39 \cdot 2^{295}$	$39 \cdot 2^{295}$
				32 (Pollard)	2^{60}	1467	$9 \cdot 2^{65}$	$18 \cdot 2^{618.5}$
FSB$_{512}$	31×2^{16}	248	1987	16	2^{393}	1962	$53 \cdot 2^{397}$	$53 \cdot 2^{397}$
				32 (Pollard)	2^{60}	1956	$12 \cdot 2^{65}$	$24 \cdot 2^{863}$

Table 7.1. Parameters of the FSB variants and estimates for the cost of generalized birthday attacks against the compression function. Storage is measured in bytes.

Clamping 60 bits in each step leaves 273 bits uncontrolled so the Pollard variant of Wagner's algorithm (see Section 2.2) becomes more efficient than the plain attack. This attack generates 16 lists of size 2^{60}, containing entries which are the xor of 5 columns from 5 distinct blocks each. This gives us the possibility to clamp 10 bits through precomputation, leaving $B = 630$ bits for each entry on level 0.

The time required by this attack is approximately 2^{224} (see (2.3)). This is substantially faster than a brute-force collision attack on the compression function, but is clearly much slower than a brute-force collision attack on the hash function, and even slower than a brute-force *preimage* attack on the hash function.

Similar statements hold for the other full-size versions of FSB. Table 7.1 gives rough estimates for the time complexity of Wagner's attack without storage restriction and with storage restricted to a few hundred exabytes (2^{60} entries per list). These estimates only consider the number and size of lists being a power of 2 and the number of bits clamped in each level being the same. The estimates ignore the time complexity of precomputation. Time is computed according to (2.2) and (2.3) with the size of level-0 entries (in bytes) as a constant factor.

Although fine-tuning the attacks might give small speedups compared to the estimates, it is clear that the compression function of FSB is oversized, assuming that Wagner's algorithm in a somewhat memory-restricted environment is the most efficient attack strategy.

References

1. MPICH2 : High-performance and widely portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/ (accessed 2009-08-18).
2. Daniel Augot, Matthieu Finiasz, Philippe Gaborit, Stéphane Manuel, and Nicolas Sendrier. SHA-3 Proposal: FSB, 2009. http://www-rocq.inria.fr/secret/CBCrypto/index.php?pg=fsb.
3. Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. A fast provably secure cryptographic hash function, 2003. http://eprint.iacr.org/2003/230.
4. Daniel Augot, Matthieu Finiasz, and Nicolas Sendrier. A family of fast syndrome based cryptographic hash functions. In Ed Dawson and Serge Vaudenay, editors, *Mycrypt*, volume 3715 of *LNCS*, pages 64–83. Springer, 2005.
5. Paulo S. L. M. Barreto and Vincent Rijmen. The WHIRLPOOL Hashing Function. http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html.
6. Daniel J. Bernstein. Better price-performance ratios for generalized birthday attacks. In *Workshop Record of SHARCS'07: Special-purpose Hardware for Attacking Cryptographic Systems*, 2007. http://cr.yp.to/papers.html#genbday.
7. Jean-Sébastien Coron and Antoine Joux. Cryptanalysis of a provably secure cryptographic hash function, 2004. http://eprint.iacr.org/2004/013.
8. Donald E. Knuth. *The Art of Computer Programming. Vol. 2, Seminumerical Algorithms.* Addison-Wesley Publishing Co., Reading, Mass., third edition, 1997. Addison-Wesley Series in Computer Science and Information Processing.
9. Lorenz Minder and Alistair Sinclair. The extended k-tree algorithm. In Claire Mathieu, editor, *SODA*, pages 586–595. SIAM, 2009.
10. Michael Naehrig, Christiane Peters, and Peter Schwabe. SHA-2 will soon retire. To appear. http://cryptojedi.org/users/peter/index.shtml#retire.
11. David Wagner. A generalized birthday problem (extended abstract). In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002: 22nd Annual International Cryptology Conference*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–304. Springer, Berlin, 2002. See also newer version [12]. http://www.cs.berkeley.edu/~daw/papers/genbday.html.
12. David Wagner. A generalized birthday problem (extended abstract) (long version), 2002. See also older version [11]. http://www.cs.berkeley.edu/~daw/papers/genbday.html.