FSBday:

Implementing Wagner's Generalized Birthday Attack against the SHA-3 Candidate FSB

Christiane Peters, Peter Schwabe
joint work with Dan Bernstein, Tanja Lange and Ruben Niederhagen

Eindhoven University of Technology

June 16, 2009

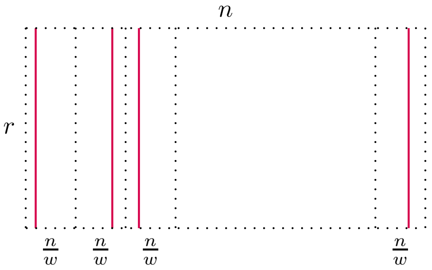
Research Retreat on Code-Based Cryptography, INRIA Rocquencourt

The compression function of FSB$_{\texttt{length}}$

Given a binary random $r \times n$ matrix H and a parameter w which indicates the number of blocks in H.

Input: a regular weight-w bit string of length n, i.e., there is exactly a single 1 in each block $[(i-1)\frac{n}{w}, i\frac{n}{w}]_{1 \le i \le w}$.

Output: Xor the w columns indicated by the input bit string.



- A collision is given by $2w$ columns — exactly two per block — which add up to 0.

Parameters

- Several parameter sets in order to satisfy NIST's requirement of having output `lengths` 160, 224, 256, 384, and 512 bits, respectively.

- SHA-3 proposal additionally includes FSB_{48}: a toy version which can be used as a training case.

Wagner's generalized birthday attack

Given 2^{i-1} lists containing B-bit strings.

Generalized birthday problem:
The 2^{i-1}-sum problem consists of finding 2^{i-1} elements — exactly one per list — such that their sum equals 0 (modulo 2).

Wagner (CRYPTO 2002)

We can expect a solution to the generalized birthday problem after one run of an algorithm using time $O((i-1) \cdot 2^{B/i})$ and lists of size $O(2^{B/i})$.

Wagner's tree algorithm

Given 4 lists containing each about $2^{B/3}$ elements which are chosen uniform at random from $\{0,1\}^B$.

- On level 0 take two lists and compare their elements on their least significant $B/3$ bits.

  Merge: If two elements coincide on those $B/3$ bits; put the xor of both elements into a new list. Proceed in the same manner with the other two lists.

  Given the uniform randomness of the elements we expect both lists to contain about $2^{B/3}$ elements.

- On level 1 take the remaining two lists. Compare their elements by considering the remaining $2B/3$ bits.

  We expect to get 1 match after the merge step.

Tree algorithm for 2^{i-1} lists

Given 2^{i-1} lists containing each about $2^{B/i}$ bit strings of length B. Suppose the bit strings were picked uniform 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.
- 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.

Tree algorithm for 2^{i-1} lists

- 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; each element is the xor of 2^{i-2} elements; the least significant $(i-2)B/i$ bits are zero.

- Comparing the elements of both lists on their $2B/i$ remaining bits gives 1 expected match.

- Each element is the xor of elements from the previous steps; it is the xor of 2^{i-1} elements and thus a solution to the generalized birthday problem.

Precomputation step

Suppose that there is space for lists of size only 2^c with $c < B/i$.

Bernstein:

- Generate $2^{c\cdot(B-ic)}$ entries and only consider those of which the least significant $B - ic$ bits are zero.
- Then apply Wagner's algorithm with lists of size 2^c and clamp away c bits on each level.

Generalization:

- The least significant $B - ic$ 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.
- If an attack does not produce a collision we simply restart the attack with different clamping values.

Repeating (parts of) the tree algorithm

- When performing the algorithm with smaller lists some bits are left "uncontrolled" at the end.
- Deal with the lower success probability by repeatedly running the attack with different clamping values.
- We can apply the same idea of changing clamping values to an arbitrary merge step of the tree algorithm.

Using Pollard iteration

- Assume that due to memory restrictions the number of uncontrolled bits is high.
- In order to find a collision of 2^{i-1} vectors we start with only 2^{i-2} lists of size $O(2^b)$ and apply the usual Wagner tree algorithm; i.e., clamp away b bits on each level.
- 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.
- This computation can be seen 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.

Expected running time

Plain Wagner:

- If we assume that the total time for one run is basically linear in the size and the number of lists and the number of levels, then the complete attack takes time

$$t = 2^{B-ib+b} = 2^{B-(i-1)b}.$$

Pollard variant:

- 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^b, so the expected running time is

$$t = 2^{B/2-(i-1)b/2+b}.$$

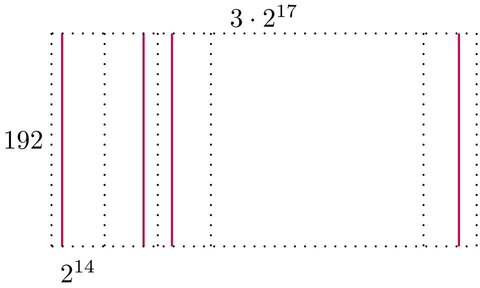
$\Longrightarrow$ Pollard variant of the attack becomes more efficient than plain Wagner with repeated runs if $B > (i+2)b$.

The compression function of FSB_{48}

Given a binary random 192×393216 matrix H; number of blocks: $w = 24$.

Input: a regular weight-24 bit string of length 393216, i.e., there is exactly a single 1 in each interval $[(i-1) \cdot 16384, i \cdot 16834]_{1 \le i \le 24}$.

Output: Xor the 48 columns indicated by the input bit string.



Goal: Find a collision in FSB_{48}'s compression function; i.e., find 48 columns — exactly 2 per block — which add up to 0.

Applying Wagner to FSB_{48}

Determine the number of lists for a Wagner attack on FSB_{48}.

- We choose 16 lists to solve this particular 48-sum problem. (16 is the highest power of 2 dividing 48).
- Each list entry will be the xor of three columns coming from one and a half blocks (no overlaps!!)

In particular:

- List $L_{0,0}$: consider sums of two columns coming from the first block of 2^{14} columns and a third column from the first half of the following block.
- We get 2^{27} sums of two columns coming from the first block. These are added to the first 2^{13} elements of the second block of the matrix H; in total roughly 2^{40} elements for $L_{0,0}$.
- List $L_{0,1}$ contains sums of columns coming from the second half of the second block and the third block. This yields again about 2^{40} possible list entries.
- Similarly, we construct the lists $L_{0,2}$, $L_{0,3}, \ldots, L_{0,15}$.

Straightforward Wagner

- The columns of H were chosen uniform at random from $\{0,1\}^{192}$.

- Assume that taking sums of those elements does not bias the distribution of 192-bit strings.

- Applying Wagner's attack with 16 lists 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.

List entries

- For each list we generate more than twice the amount needed for a straightforward attack.
- 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 2 bits away should thus yield a list of 2^{38} bit strings.
- Now we ignore those 2 least significant bits which are 0 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 list entries

Technische Universiteit Eindhoven University of Technology

- List entries could be 192-bit strings; namely the sums of columns of H.
- We don't need to store the whole bit string; bits we already know to be 0 do not have to be stored; so in each level of the tree the number of bits per entry decreases.
- However, we know that a successful attack will produce a list containing the all-zero bit string at the end.
- In order to identify a collision we have to store the column positions in the matrix that lead to this all-zero value.
- Unlike storage requirements for values the number of bytes for positions increases with increasing levels.

Storing positions

Technische Universiteit Eindhoven University of Technology

- Dynamic recomputation reduces the storage requirements by not storing the entry value at all but recomputing it every time it is needed from the positions.
- There are 2^{40} possibilities to choose columns to produce entries of a list, so we can encode the positions in 40 bits (5 bytes).
- 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.

What list size can we handle?

- We start with 16 lists of size 2^{38}, each containing bit strings of length $r' = 190$.
- We store the column positions of each entry which we encode in 40 bits (5 bytes).
- Storing 16 lists with 2^{38} entries, each entry encoded in 5 bytes requires 20480 GB of storage space.
- The Coding and Cryptography Computer Cluster at Eindhoven University of Technology only has a total hard disk space of 7 TB, so we have to adapt our attack strategy to this limitation.

Adapt attack strategy

Technische Universiteit **Eindhoven** University of Technology

- On the first level we have 16 lists and as we need at least 5 bytes per list entry we can handle at most $7 \cdot 2^{40}/2^4/5 = 1.36 \times 2^{36}$ entries per list.
- A straightforward implementation would use lists of size 2^{36}: consider 2^{40} entries per list and clamp 4 bits during list generation; this leads to 2^{36} values for each of the 16 lists.
- These values have a length of 188 bits represented by 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 unknown bits.
- We expect to run the attack 256.5 times until we find a collision.

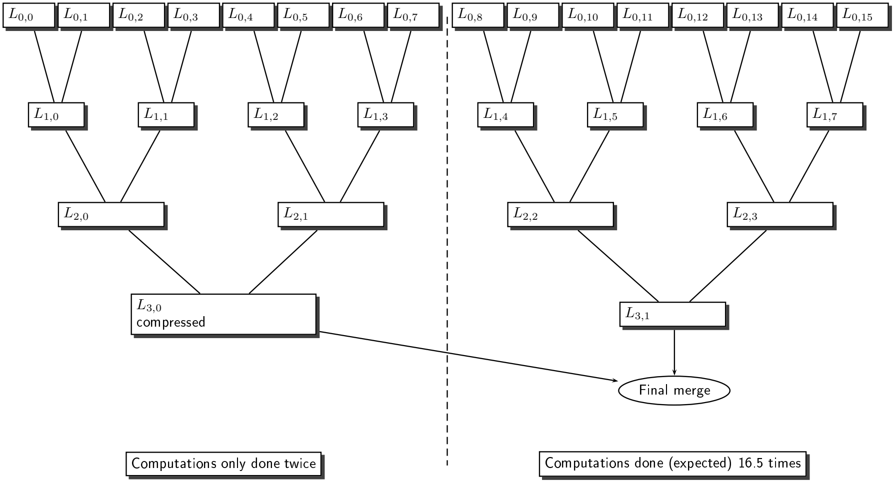
Half-tree compression

- First compute left half-tree, using 8 lists of size 2^{37} (5 TB)
- Clamp 3 bits through precomputation
- Resulting list $L_{3,0}$ has entries with $189 - 3 \cdot 37 = 78$ remaining bits
- Now save values instead of positions, compression by factor of 4 (1.25 TB)
- Compute right half-tree (5 TB, total of 6.25 TB) and perform last merge
- In case of collision: Compute left half-tree again to reconstruct positions

Half-tree compression

- First compute left half-tree, using 8 lists of size 2^{37} (5 TB)
- Clamp 3 bits through precomputation
- Resulting list $L_{3,0}$ has entries with $189 - 3 \cdot 37 = 78$ remaining bits
- Now save values instead of positions, compression by factor of 4 (1.25 TB)
- Compute right half-tree (5 TB, total of 6.25 TB) and perform last merge
- In case of collision: Compute left half-tree again to reconstruct positions
- Otherwise: Change clamping constants in right half-tree
- Expected: 18.5 half-tree computations ($2\times$ left half-tree, $16.5\times$ right half-tree)

Attack Strategy



- Coding and Cryptography Computer Cluster
- 10 machines, each equipped with
  - Intel Core 2 Quad Q6600 processor (2.4 GHz),
  - 8 GB of RAM supporting ECC,
  - Marvell PCI-E Gigabit Ethernet cards,
  - Western Digital 700 GB SATA hard disk.
- For this project: Communication through MPI (MPICH2)
  - Offers synchronous message based communication
  - Standard for HPC applications
  - MPICH2 provides an ethernet back-end

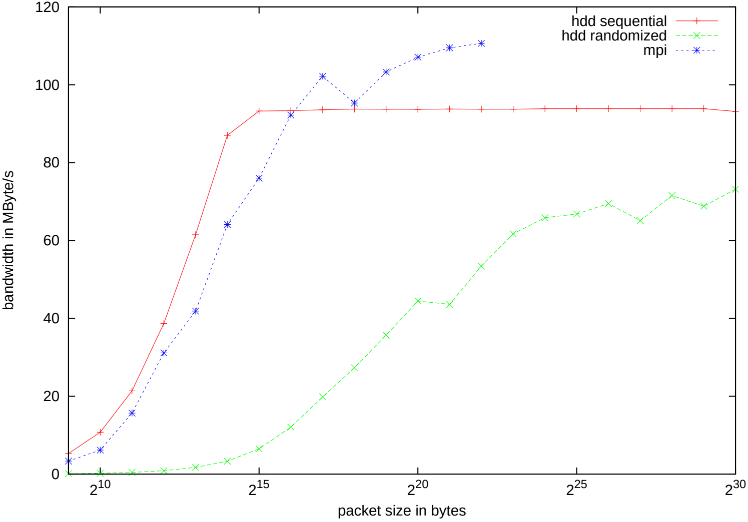
Finding the bottleneck(s)

- Basically every byte needs to be stored, sent, and loaded 4 times.
- Possible performance bottlenecks
  - CPU computation power
  - Network throughput
  - Hard-disk throughput

Finding the bottleneck(s)

- Basically every byte needs to be stored, sent, and loaded 4 times.
- Possible performance bottlenecks
  - CPU computation power
  - Network throughput
  - Hard-disk throughput
- If the CPU is too slow we have to write faster code
- Determine network throughput: IBM MPI benchmark
- Determine hard-disk throughput: our own hard-disk benchmark
  - Direct I/O, no filesystem
  - Sequential and randomized access patterns

Finding the bottleneck(s)



Finding the bottleneck(s)

- Basically every byte needs to be stored, sent, and loaded 4 times.
- Possible performance bottlenecks
  - CPU computation power
  - Network throughput
  - Hard-disk throughput
- If the CPU is too slow we have to write faster code
- Determine network throughput: IBM MPI benchmark
- Determine hard-disk throughput: our own hard-disk benchmark
  - Direct I/O, no filesystem
  - Sequential and randomized access patterns

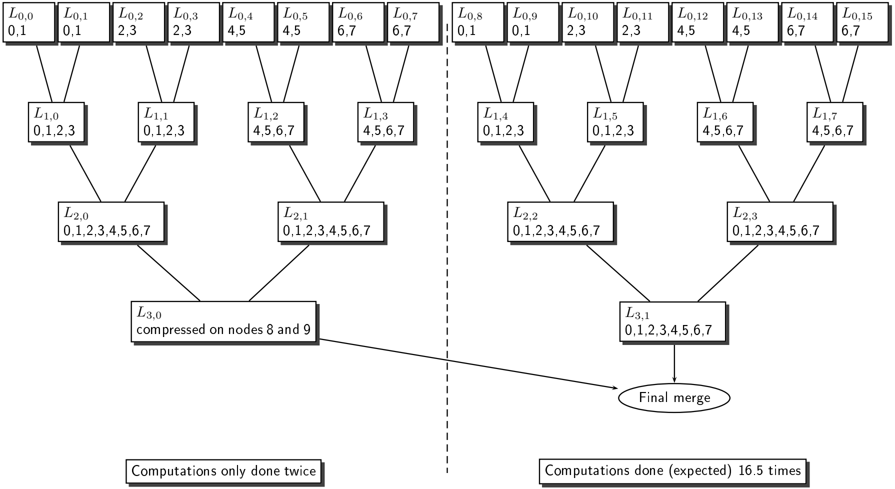
$\Longrightarrow$ Mainly bottlenecked by hard-disk throughput

Parallelization

TU/e Technische Universiteit Eindhoven University of Technology

- Distribute fractions of lists to nodes according to some of the bits relevant for sorting and merging on the next level
- Each node on each level holds two fractions of two lists
- Each node performs sort-and-merge on its list fractions

Parallelization



Parallelization

- Distribute fractions of lists to nodes according to some of the bits relevant for sorting and merging on the next level
- Each node on each level holds two fractions of two lists
- Each node performs sort-and-merge on its list fractions
- Split fractions further into 512 parts of 640 MB each (presort)
- Sort and merge parts independently in memory

Parallelization

- Distribute fractions of lists to nodes according to some of the bits relevant for sorting and merging on the next level
- Each node on each level holds two fractions of two lists
- Each node performs sort-and-merge on its list fractions
- Split fractions further into 512 parts of 640 MB each (presort)
- Sort and merge parts independently in memory
- Pipeline
  - Loading from hard disk into memory,
  - Sorting of two parts,
  - Merging of previously sorted parts,
- Requires 6 parts in memory at the same time (3.75 GB)

Parallelization

- Distribute fractions of lists to nodes according to some of the bits relevant for sorting and merging on the next level
- Each node on each level holds two fractions of two lists
- Each node performs sort-and-merge on its list fractions
- Split fractions further into 512 parts of 640 MB each (presort)
- Sort and merge parts independently in memory
- Pipeline
  - Loading from hard disk into memory,
  - Sorting of two parts,
  - Merging of previously sorted parts,
- Requires 6 parts in memory at the same time (3.75 GB)
- Two blocks of operations:
  - Load, Sort, Merge, Send
  - Receive, Presort, Store

Ales instead of Files

Technische Universiteit **Eindhoven** University of Technology

- Each node uses a large data partition `/dev/sda1`
- Opened with `O_DIRECT` (without caching)
- Organize data in chunks of 1.25 MB ("ales"), each belonging to
  - one of two list fractions,
  - one of 512 parts (per list fraction),
  - OR free space.
- AleSystem also stores number of elements per part
- Throughput with sequential access (during list generation): ~90 MB/sec (non duplex)
- Throughput with random access: ~40 MB/sec (non duplex)

Timing Results

- Compression and last merge step not (fully) implemented, yet
- Current benchmarks: One half-tree computation takes $\sim$ 33 h
  - 2:32 h for list generation
  - 9:43 h for first sort/merge step
  - 10:02 h for second sort/merge step
  - 10:46 h for third sort/merge step
- Expected: 18.5 half-tree computations: 610:30 h
- 16.5 last merge steps (estimated 12 h each): 198 h
- Expected total time: 808.5 h or 33 days and 16.5 hours

Scalability Analysis I

Wagner against FSB_{160}

- 16 lists of size 2^{127}
- Entries are xors of 10 columns from 5 blocks (2^{135}) possibilities
- Each entry requires 135 bits (17 bytes)
- Clamp 8 bits through precomputation
- Running time 2^{127} (not considering costs for precomputation)

Scalability Analysis I

Wagner against FSB_{160}

- 16 lists of size 2^{127}
- Entries are xors of 10 columns from 5 blocks (2^{135}) possibilities
- Each entry requires 135 bits (17 bytes)
- Clamp 8 bits through precomputation
- Running time 2^{127} (not considering costs for precomputation)
- Memory requirement: $17 \cdot 2^{57}$ Exabytes
- Currently available: Storage systems with a few petabytes

Scalability Analysis I

Wagner against FSB_{160}

- 16 lists of size 2^{127}
- Entries are xors of 10 columns from 5 blocks (2^{135}) possibilities
- Each entry requires 135 bits (17 bytes)
- Clamp 8 bits through precomputation
- Running time 2^{127} (not considering costs for precomputation)
- Memory requirement: $17 \cdot 2^{57}$ Exabytes
- Currently available: Storage systems with a few petabytes
- With just a few exabytes, Pollard variant becomes more efficient
- E.g. with 144 exabytes: time 2^{220}

Scalability Analysis II

Overview of Wagner against FSB variants

	Number of lists	lists	Storage (EB)	Time
FSB_{160}	16	2^{127}	$17 \cdot 2^{51}$	2^{127}
	16 (Pollard)	2^{60}	$9 \cdot 2^4 = 144$	2^{220}
FSB_{224}	16	2^{177}	$24 \cdot 2^{121}$	2^{177}
	16 (Pollard)	2^{60}	$13 \cdot 2^4 = 208$	2^{339}
FSB_{256}	16	2^{202}	$27 \cdot 2^{146}$	2^{202}
	16 (Pollard)	2^{60}	$14 \cdot 2^4 = 224$	2^{382}
	32 (Pollard)	2^{56}	18	2^{400}
FSB_{384}	16	2^{291}	$39 \cdot 2^{235}$	2^{291}
	32 (Pollard)	2^{60}	$9 \cdot 2^5 = 288$	$2^{613.5}$
FSB_{512}	16	2^{393}	$53 \cdot 2^{337}$	2^{393}
	32 (Pollard)	2^{60}	$12 \cdot 2^5 = 384$	2^{858}

Further information

Technische Universiteit **Eindhoven** University of Technology

Paper: http://cryptojedi.org/users/peter/#fsbday

Cluster: http://www.win.tue.nl/cccc/

Code: Will be available (public domain)