FSBday:
Implementing Wagner’s Generalized Birthday Attack against the round-1 SHA-3 Candidate FSB

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

Eindhoven University of Technology

September 10, 2009

SHARCS 2009
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 ’02)
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.

  Uniform randomness of the elements $\Rightarrow$ both lists will contain about $2^{B/3}$ elements.

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

  Expect to get 1 match after the merge step.
Tree algorithm for $2^{i-1}$ lists

The tree algorithm generalizes to $2^{i-1}$ lists as follows:

- Compare lists — always two at a time — by looking at the least significant $B/i$ bits of elements.

- On level $i - 2$ we are left with two lists whose elements need to be compared on $2B/i$ remaining bits.
Precomputation step

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

Bernstein (SHARCS ’07):

- 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.

Wagner

\[
\begin{array}{cccccc}
B/i & B/i & B/i & \cdots & B/i \\
\end{array}
\]

Bernstein

\[
\begin{array}{cccccc}
c & c & c & \cdots & B - ic \\
\end{array}
\]
Precomputation step

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

**Bernstein (SHARCS '07):**

- 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

- **Clamping value** does not have to be the same on all lists (but: sum of all clamping values has to be 0).

- 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.
Target: the compression function of FSB\textsubscript{48}

Given a binary random $192 \times 393216$ 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 $[(i - 1) \cdot 16384, i \cdot 16834]_{1 \leq i \leq 24}$.

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

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

Determine the number of lists for a Wagner attack on FSB\textsubscript{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!)

Straightforward Wagner

- 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.

- Reduce amount of data by clamping away 2 bits $\Rightarrow 2^{38}$ entries per list (clamp 38 bits on each level)

- Ultimately we are not interested in the value of the entry; but in the column positions in the matrix that lead to this all-zero value.
  - Value-only representation
  - Positions-only representation: keep full positions; if we need the value (or parts of it) it can be dynamically recomputed from the positions.

- Note: Unlike storage requirements for values the number of bytes for positions increases with increasing levels.
Storing positions

- Encode column positions of each entry in 40 bits (5 bytes) for the first level.

- 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.

- 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 about 5440 GB, so we have to adapt our attack strategy to this limitation.
Adapt attack strategy

- Can handle at most $5 \cdot 2^{40}/2^4/5 = 2^{36}$ entries per list.

- A straightforward implementation would use lists of size $2^{36}$: clamp 4 bits during list generation; this leads to $2^{36}$ values for each of the 16 lists.

- We expect to run the attack 256.5 times until we find a collision.
Attack in two phases

Idea

- First phase: Figure out which clamping constants yield collision
- Second phase: Compute matrix positions yielding collision
- During phase one we do not have to store positions of entries
- On each level compress entries to shortest possible representation
  - Level 0: 5 bytes (positions only)
  - Level 1: 10 bytes (positions only)
  - Level 2: 13 bytes (values only)
  - Level 3: 9 bytes (values only)
- Use lists of size $2^{37}$
- Clamp 3 bits through precomputation
- This leaves 4 bits “uncontrolled”
Attack Strategy

\[ \Rightarrow 1152 \text{ GB} + 1664 \text{ GB} + 2560 \text{ GB} = 5376 \text{ GB} \]
Our Strategy

- Continue the computation with different clamping constants until $L_{4,0}$ contains at least one entry
- Store the values in $L_{3,0}$ and $L_{3,1}$ that yield the collision
- Recompute $L_{3,0}$ and $L_{3,1}$ using positions-only representation to find positions in the matrix
- Expected:
  - $1 \times$ Computation of $L_{3,0}$ (values only)
  - $1 \times$ Computation of $L_{2,2}$ (values only)
  - $16.5 \times$ Computation of $L_{2,3}$, $L_{3,1}$, $L_{4,0}$ (values only)
  - $1 \times$ Computation of $L_{3,0}$ (positions only)
  - $1 \times$ Computation of $L_{3,1}$ (positions only)
Finding the bottleneck(s)

- Basically every byte needs to be stored, sent, and loaded.
- 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.
- 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)

![Graph showing bandwidth in MByte/s against packet size in bytes for hdd sequential, hdd randomized, and mpi.]
Finding the bottleneck(s)

- Basically every byte needs to be stored, sent, and loaded.
- 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

⇒ Mainly bottlenecked by hard-disk throughput
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
Parallelization
Parallelization

- Split fractions further into 512 parts of 640 MB each (presort, according to 9 bits)
- 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

- Split fractions further into 512 parts of 640 MB each (presort, according to 9 bits)
- 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
Parallelization

- Split fractions further into 512 parts of 640 MB each (presort, according to 9 bits)
- 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
- Observe: Bits known through node and presorting do not have to be stored in value-only representation
- That’s how we get down to 13 and 9 bytes on levels 2 and 3 respectively
Ales instead of Files

- Each node uses a large data partition `/dev/sda1`
- Opened with `O_DIRECT` (without caching)
- Organize data in chunks of 1,198,080 Bytes ("ales")
- This value is a multiple of 9, 13, 40 (entry sizes) and 512 (for DMA)
- AleSystem also stores number of elements per part
- Throughput with sequential access (during list generation): \(~90\) MB/sec
- Throughput with random access: \(~40\) MB/sec
Timing Results

- Current benchmarks for phase 1:
  - Computation of list $L_{3,0}$: $\sim 32$ h (once)
  - Computation of list $L_{2,2}$: $\sim 14$ h (once)
  - Computation of list $L_{2,3}$: $\sim 14$ h (exp. $16.5 \times$)
  - Computation of list $L_{3,1}$: $\sim 4$ h (exp. $16.5 \times$)
  - Check for collision in $L_{3,0}$ and $L_{3,1}$: $\sim 3.5$ h (exp. $16.5 \times$)

- Expected time for phase 1: $32 + 14 + 16.5 \cdot (14 + 4 + 3.5) = 400.7$ h or 17 days

- Time for phase 2: $\sim 33$ h per half-tree, in total $\sim 66$ h

- Expected time in total: $\sim 19.5$ days.
Timing Results

- Current benchmarks for phase 1:
  - Computation of list $L_{3,0}$: $\sim 32$ h (once)
  - Computation of list $L_{2,2}$: $\sim 14$ h (once)
  - Computation of list $L_{2,3}$: $\sim 14$ h (exp. 16.5×)
  - Computation of list $L_{3,1}$: $\sim 4$ h (exp. 16.5×)
  - Check for collision in $L_{3,0}$ and $L_{3,1}$: $\sim 3.5$ h (exp. 16.5×)

- Expected time for phase 1: $32 + 14 + 16.5 \cdot (14 + 4 + 3.5) = 400.7$ h or 17 days

- Time for phase 2: $\sim 33$ h per half-tree, in total $\sim 66$ h

- Expected time in total: $\sim 19.5$ days.

- Some parts of the code might be optimized further

- The attack is stateful so it is easy to exchange code with faster version
Further information


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

Code: Will be available (public domain)