An Improved Algebraic Attack on Hamsi-256

Itai Dinur and Adi Shamir

Computer Science department
The Weizmann Institute
Rehovot 76100, Israel

Abstract. Hamsi is one of the 14 second-stage candidates in NIST's SHA-3 competition. The only previous attack on this hash function was a very marginal attack on its 256-bit version published by Thomas Fuhr at Asiacrypt 2010, which is better than generic attacks only for very short messages of fewer than 100 32-bit blocks, and is only 26 times faster than a straightforward exhaustive search attack. In this paper we describe a different algebraic attack which is less marginal: It is better than the best known generic attack for all practical message sizes (up to 4 gigabytes), and it outperforms exhaustive search by a factor of at least 512. The attack is based on the observation that in order to discard a possible second preimage, it suffices to show that one of its hashed output bits is wrong. Since the output bits of the compression function of Hamsi-256 can be described by low degree polynomials, it is actually faster to compute a small number of output bits by a fast polynomial evaluation technique rather than via the official algorithm.

Keywords: Algebraic attacks, second preimages, hash functions, Hamsi.

1 Introduction

The Hamsi family of hash functions [1] was designed by Özgül Küçük and submitted to the SHA-3 competition in 2008. In 2009 it was selected as one of the 14 second round candidates of the competition. Hamsi has two instances, Hamsi-256 and Hamsi-512, that support four output sizes 224, 256, 384 and 512.

Previous results on Hamsi include distinguishers [4] and [5], pseudo-preimage attacks [6] and near collision attacks [7]. However, these results do not break the core security properties of a hash function. More recently, Thomas Fuhr introduced the first real attack on Hamsi-256 [2]. The attack exploits linear relations between some input bits and output bits of the compression function in order to find pseudo preimages for the compression function of Hamsi-256 (a pseudo preimage of an arbitrary chaining value h_i^* under the compression function $\mathcal{F}$ is a message block $\bar{M}_i$ and a chaining value $\bar{h}_{i-1}$ such that $\mathcal{F}(\bar{M}_i, \bar{h}_{i-1}) = h_i^*$). The pseudo preimages can then be used in order to find a second preimage for a given message with complexity $2^{251.3}$, which is better than exhaustive search by a marginal factor of $2^{4.7} \approx 26$ (whose existence and exact size depends on how we measure the complexity of various operations). In addition, Fuhr's attack is better than a generic long message attack only for very short messages with up to 96 32-bit blocks[1](i.e. 384 bytes). Nevertheless, it is the first attack on Hamsi-256 that violates its core security claims.

In this paper, we develop new second preimage attacks on Hamsi-256 which are slightly less marginal. Our best attack on short messages of Hamsi-256 runs in time which is faster than exhaustive search by a factor of 512, which is about 20 times faster than Fuhr's attack. For longer messages, we develop another attack which is faster than the Kelsey and Schneier attack by a factor which is between 6 and 4 for all messages of practical size (i.e., up to 4 gigabytes). Our short message attack exploits some of the observations made in [2] regarding the Hamsi Sbox, but uses them in a completely different way to obtain better results: While Fuhr solved linear equations in order to speed up the search for pseudo preimages, our attacks use fast polynomial enumeration algorithms to quickly discard compression function inputs which cannot possibly yield the desired output.

Since the straightforward evaluation of the compression function of Hamsi-256, Fuhr's attack, and our attacks use different bitwise operations, comparing these attacks on Hamsi-256 cannot be done simply by counting the number of compression function evaluations. Instead, we compare the complexity of straightline implementations of the algorithms, counting the number of bit operations (such as AND, OR, XOR) on pairs of bits and ignoring bookkeeping operations such as moving a bit from one position to another (which only requires renaming of variables in straightline programs). In this model of computation, the best available implementation of one compression function evaluation of Hamsi-256 (given as part of the submission package in [1] and used as the reference complexity in this paper), requires about $10,500$ bit operations. Our best attack is about 512 times faster, and is thus equivalent to an algorithm than performs only 20 bit operations per message block.

Polynomial enumeration algorithms evaluate a polynomial function over all its possible inputs. Clearly, the complexity of such enumeration algorithms must be at least 2^n for n-bit functions and thus they may seem to provide little advantage over trivial exhaustive search. However, cryptographic primitives are usually heavy algorithms that require substantial computational effort per execution. Consequently, the complexity of exhaustive search (measured by the number of bit operations) can be much higher than 2^n. However, for low degree polynomials, the complexity of enumeration algorithms is higher than 2^n only by a small multiplicative factor. In order to attack Hamsi-256, we search for polynomials of low degree that relate some of the bits computed by the compression function: The variables of each polynomial are chosen from the inputs to Hamsi-256, and the output of each polynomial is either an output bit of Hamsi-256, a linear combination of output bits of Hamsi-256, or an intermediate state bit of Hamsi-256 from which an output bit (or output bits) can be easily computed.

[1] Since Hamsi-256 is built using the Merkle-Damgård construction and it has a 256-bit intermediate state, the best known generic second preimage attack on Hamsi-256 with long messages is the Kelsey and Schneier attack [3] that runs in time $k \cdot 2^{128} + 2^{256-k}$ for messages of length 2^k.

Our attack on short messages of Hamsi-256 is divided into two stages: In the first stage we find multiple pseudo preimages of a single target chaining value obtained by one of the invocations of the compression function during the computation of the hash of the given message. In the second stage we obtain a second preimage for the message by searching for a second preimage for one of the target pseudo preimages that are found in the first stage (this is done by traversing a tree-like structure of chaining values, as shown in figure 1). In both stages, we first efficiently enumerate a set of low degree polynomials for all the possible values of a carefully chosen set of variables which are input to Hamsi-256. We then run the compression function only for the inputs for which the polynomial evaluations match the values of the target (or targets). Since the compression function of Hamsi-256 mixes the chaining value less extensively than the message, in the first stage we find only pseudo preimages by selecting our set of input variables of the enumerated polynomials among the bits of the chaining value. In the second stage, we have to find second preimages and thus we have to select our set of input variables of the enumerated polynomials among the message bits. Therefore, the polynomials enumerated in the first stage have a lower degree than those enumerated in the second stage, implying that the first stage gives a better improvement factor than the second stage (compared to exhaustive search). We note that our two-stage process of finding a second preimage using an efficient pseudo preimage search algorithm is a variant of the well-known meet-in-the-middle algorithm, described in appendix B. The difference is that the second stage of meet-in-the-middle is performed using exhaustive search, whereas the second stage of our algorithm is optimized using efficient polynomial enumeration algorithms.

For longer messages, the generic attack of Kelsey and Schneier becomes increasingly better with the length, and quickly overperforms both Fuhr's attack and our enumeration-based attack. In this case, we develop another attack that directly plugs into and speeds up the algorithm of Kelsey and Schneier. The attack is based on the second stage of our short message attack, but uses different parameters since in this case we try to find a second preimage for a potentially huge number of targets.

The fact that our short message attack is faster than Fuhr's attack may seem surprising, as Fuhr's attack is based on very simple and efficient algorithms for solving linear equations, whereas our attack is based on exponential-time polynomial enumeration algorithms. However, linear equations are much more difficult to obtain than non-linear equations of relatively low degree. In particular, Fuhr can obtain useful linear equations in only 7 or 8 variables in the first stage. The complexity of interpolating and solving such a system is faster than exhaustive search (which requires 2^7 or 2^8 function evaluations) only by a small factor. In the second stage, [2] can not obtain any linear equations and proceeds by performing an exhaustive search, which makes the attack faster than Kelsey and Schneier's attack only for very short messages. Another reason why our attack is faster is that in the first stage we also exploit the weak diffusion of the input variables into some of the output bits. This allows our enumeration algorithms to evaluate some polynomials only over the possible values of small subsets of variables in order to obtain the values for the entire variable set.

2 Description of Hamsi-256

In this section we provide a brief description of the compression function of Hamsi-256. For more details, please refer to its specification [1].

The compression function of Hamsi-256 takes as an input a 32-bit message block M_i and a 256-bit chaining value h_i and outputs a new 256-bit chaining value h_{i+1}. The compression function first expands the 32-bit message to 8 blocks of 32 bits using a linear code over $GF(4)$: $E(M_i) = (m_0, m_1, ..., m_7)$. The expanded message is then concatenated to the chaining value to form a 512-bit state treated as a 4×4 matrix of 32-bit blocks as follows:

$$(m_0, m_1, ..., m_7, c_0, c_1, ..., c_7) \longrightarrow \begin{array}{|c|c|c|c|} \hline s_0 & s_1 & s_2 & s_3 \\ \hline s_4 & s_5 & s_6 & s_7 \\ \hline s_8 & s_9 & s_{10} & s_{11} \\ \hline s_{12} & s_{13} & s_{14} & s_{15} \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline m_0 & m_1 & c_0 & c_1 \\ \hline c_2 & c_3 & m_2 & m_3 \\ \hline m_4 & m_5 & c_4 & c_5 \\ \hline c_6 & c_7 & m_6 & m_7 \\ \hline \end{array}$$

The concatenation is followed by a permutation defined by three rounds, where each round consists of three layers: In the first layer, the state bits are XORed with some constants. In the second layer, the 128 4-bit columns of the state undergo simultaneous applications of a 4×4 Sbox described in table 1.

Table 1. The Hamsi Sbox.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	8	6	7	9	3	C	A	F	D	1	E	4	0	B	5	2

The third layer consists of several parallel applications of a linear transformation L on the state.

$$(s_0, s_5, s_{10}, s_{15}) := L(s_0, s_5, s_{10}, s_{15})$$
$$(s_1, s_6, s_{11}, s_{12}) := L(s_1, s_6, s_{11}, s_{12})$$
$$(s_2, s_7, s_8, s_{13}) := L(s_2, s_7, s_8, s_{13})$$
$$(s_3, s_4, s_9, s_{14}) := L(s_3, s_4, s_9, s_{14})$$

Finally, the second and fourth rows of the state are discarded and the initial chaining value h_i is XORed with the remaining state to form h_{i+1}. The last message block is processed differently, by applying 8 rounds of this permutation 8 (instead of the standard 3).

3 A Direct Attack on Hamsi-256

3.1 The Properties and Weaknesses of Hamsi-256 which are Exploited by the Attack

In the direct attack our goal is to consider the 32 message bits as variables and the 256-bit chaining value as a fixed input and analyze the degree of the state bits after each one of the three rounds of the Hamsi-256 compression function as polynomials in the message bits. Every Hamsi Sbox can be described as a polynomial of degree 3 in its 4 input variables. However, due to the way the expanded message bits and the chaining value bits are interleaved in Hamsi-256, after 1 round of the compression function, each state bit is a polynomial of reduced degree 2 in the message bits. This may seem insignificant, but after 2 rounds, the degree of each state bit as a polynomial in the message is at most 6 instead of the expected value of $3^2 = 9$. After the final compression function round, the degree is 18 instead of the expected 27. This low algebraic degree can be used to obtain distinguishers on Hamsi-256 (as already noticed in [4] and [5]), but even this reduced degree is too high for our algebraic attack. Instead, we exploit the low diffusion property of one round of Hamsi-256, namely, that several output bits of the compression function depend only on a small number of inputs from the second round.

3.2 Analysis of Polynomials of Degree 6 in 32 Variables

Since the attack on Hamsi-256 relies on a slightly improved version of exhaustive search, we have to use every possible saving and shortcut in the implementation of our algorithms, and can not ignore constants or low-order terms. In particular, we show how to efficiently interpolate and evaluate any polynomial of degree 6 in 32 variables for all the 2^{32} possible values of its inputs using fewer than $7 \cdot 2^{32}$ bit operations (instead of the 2^{64} complexity of the naive evaluation of the 2^{32} possible terms for each one of the 2^{32} possible inputs):

1. Given any black box implementation of the polynomial (e.g. in the form of the Hamsi-256 program), evaluate the output of the polynomial (which is a single bit value) only for input vectors of hamming weight ≤ 6 and store all the results.
2. Compute the coefficient of each term t_I of degree at most 6, where $I \subset \{0, 1, ...31\}, |I| \leq 6$ represents some subset of variables multiplied together. The coefficient of t_I is computed by summing all the outputs of the polynomial obtained from all inputs which assign 0 values to all variables that are not contained in I (where the variables that are contained in I are assigned all possible values).
3. Allocate an array of 2^{32} bits and copy all the coefficients of the polynomial into the array. The coefficient t_I is copied into the entry whose binary index is encoded by $b_0, b_1, ..., b_{31}$ where $b_i = 1$ if and only if $i \in I$. All the other entries of the array are set to 0.

4. Apply the Moebius transform [8] on the array and obtain an array which contains the evaluations of the polynomial for all 2^{32} possible input values.

Step 1 requires $\sum_{i=0}^{6} \binom{32}{i} \approx 2^{20}$ compression function evaluations. Step 2 requires $\sum_{i=0}^{6} 2^i \binom{32}{i} < 2^{26}$ bit operations. Step 3 can be combined with step 2 by writing the coefficients directly into the array and does not require additional work. A naive implementation of step 4 requires $32 \cdot 2^{31}$ bit operations, since the generic Moebius transform consists of 32 iterations, where in each iteration we add half of the array entries to the other half. However, in our case, the initial array can contain only about 2^{20} non zero values, whose locations are known and represent all the vectors with hamming weight of at most 6. In the first iteration of the Moebius transform, we split the array into 2 parts according to one variable and add only the entries whose index has a hamming weight of at most 6 in the remaining 31 variables (the others entries are left unchanged). The total complexity of the first iteration is thus $\sum_{i=0}^{6} \binom{31}{i}$ bit operations. In the second iteration, each half of the array is split into 2 parts according to another variable. Similarly, we add only the entries whose index has a hamming weight of at most 6 in the remaining 30 variables. The total complexity of the second iteration is thus $2 \sum_{i=0}^{6} \binom{30}{i}$. Generally, the complexity of the $j'th$ iteration where $0 \leq j \leq 25$ is $min(2^{31}, 2^j \sum_{i=0}^{6} \binom{31-j}{i})$. For $26 \leq j \leq 31$, the complexity is 2^{31}. Summing over all iterations, we get a total complexity of less than $7 \cdot 2^{32}$. We note that it is also possible to use the Gray-code based polynomial enumeration algorithm recently presented in [9] which is a bit more complicated than our Moebius-based transform, and has a similar time complexity.

Assuming that the straightforward evaluation of one Hamsi-256 compression function requires about $10,500$ bit operations, step 4 is the heaviest step and dominates the complexity of the algorithm. Note that when analyzing several polynomials which correspond to different output bits, step 1 needs to be performed only once since every compression function evaluation gives us the values of all the required polynomials. In addition, the bit locations XORed together during the Moebius transformation do not depend on the evaluated polynomial, and thus the evaluation of k unrelated polynomials can be achieved by XORing k bit words instead of single bits. This is particularly convenient when $k = 32$ or $k = 64$, which are standard word sizes in modern microprocessors.

3.3 Efficiently Eliminating Wrong Messages

Assume that we are given a target chaining value h_i^*, and a fixed chaining value h_{i-1}. We would like to efficiently find a single message block M_i such that

$\mathcal{F}(M_i, h_{i-1}) = h_i^*$, or decide that such a message does not exist for the given h_{i-1}, h_i^*. Due to the short message blocks of Hamsi-256, the probability to find a desired message for random chaining values h_{i-1} and h_i^* is about $2^{32-256} = 2^{-224}$. Hence, to succeed with high probability for a given h_i^*, we have to generate about 2^{224} random values for h_{i-1}. In order to obtain this number of chaining values, the target h_i^* must be located at least in block number 8 of the message (i.e. $i \geq 8$). This implies that we can apply our attack only when the given message contains at least 8 blocks.

The idea is to algebraically compute only a small set of output bits (indexed by N) for all the 2^{32} messages and compare their values to the values of those bits of the target chaining value. If the bits match, we run the compression function for the message to compute the whole 256-bit output, and compare it to the target h_i^*. Otherwise, we discard the message. Using the algorithm of section 3.2, we efficiently evaluate only the bits produced after 2 rounds of the compression function, which are required in order to determine the output bits specified by N. We then combine these values as specified by the last round, and obtain the values of the output bits indexed by N for all the possible 2^{32} messages.

In order to minimize the complexity of the attack, we need to select a set N that is big enough to eliminate a large number of messages with high probability. On the other hand, choosing N too big, will force us to evaluate many bits after two rounds, increasing the complexity of the attack. In fact we don't need to analyze all the second round bits that are required to compute the output bits N: We write the ANF form of the output bits N as a function of the second round bits and note that the sum of the second round variables which are not multiplied together (which we call the simple sum) is itself a polynomial of degree 6 in the message. Thus, the simple sum of each such output bit can be analyzed in the same way as the second round bits without computing separately each one of the summed bits. Note that in Step 1 of the analysis (interpolation), evaluating the polynomial means computing the sum of variables numerically from the output. The complexity of this computation is negligible compared to a compression function evaluation of Hamsi-256.

After choosing the set N of output bits and the set of second round bits of degree 6 $S(N)$ we have to evaluate, we do the following:

1. Given that $i \geq 8$: Choose an arbitrary message prefix of $i-8$ blocks $M_1, M_2, ..., M_{i-8}$ and compute $h_{i-8} = \mathcal{F}(M_1, M_2, ..., M_{i-8}, IV)$ (which is fixed throughout the attack). Use DFS to traverse the tree of chaining values rooted at h_{i-8} by assigning values to the blocks $M_{i-7}, M_{i-6}, ..., M_{i-1}$ and computing the next chaining value h_{i-1} (as shown in figure 1). For each generated h_{i-1}:
2. Evaluate all the bits of $S(N)$ for all possible 32-bit message values using the algorithm of Section 3.2 and list them as $|S(N)|$ bit arrays of size 2^{32}.
3. Using the ANF form of the outputs bits of N as a function of the second round bits, calculate $|N|$ bit arrays of size 2^{32} representing the values of the $|N|$ output bits for all the messages.

4. Traverse the $|N|$ bit arrays and check whether the values of the output bits in N match the values of those bits in h_i^*, for each message M_i. Store all the messages for which there is a match.
5. For each message M_i stored in the previous step, evaluate the full Hamsi-256 compression function output by using its standard implementation and check whether $\mathcal{F}(M_i, h_{i-1}) = h_i^*$. If equality holds, output $M = M_1, M_2, ..., M_i$. Otherwise go to step 1.

The memory requirements of the algorithm can be reduced by calculating the bits of N iteratively and eliminating wrong messages according to the current calculated bit. We can then reuse some memory which was used for the calculation of the previous bits of $S(N)$ and which is not required anymore.

Given that $|N| = n_1$, $|S(N)| = n_2$, and the number of bit operations per message that is required to compute N from $S(N)$ is n_3 (calculated using the ANF form of the outputs bits), the complexity of the attack is about $2^{224}(n_3 2^{32} + 7n_2 2^{32} + 10500 \cdot 2^{32-n_1}) =$
$2^{256}(n_3 + 7n_2 + 10500 \cdot 2^{-n_1})$ bit operations. Compared to exhaustive search which requires about $10500 \cdot 2^{256}$ bit operations, this gives an improvement factor of about $(\frac{n_3+7n_2}{10500} + 2^{-n_1})^{-1}$.

We searched for sets of output bits N that optimize the complexity of the attack. The best set that we found is $N = \{5, 156, 184, 214, 221, 249\}$ whose 6 output bits depend just on 56 second round bits. We also have to add 6 bits for the simple sums of the second round variables, and the full list of $56+6 = 62$ bits is described in appendix A. For this parameter set we get that $n_2 = 62$, $n_1 = 6$ and the computation of all the evaluated output bits requires about $n_3 = 150$ bit operations per message. We calculated this number after a few optimizations which are based on finding some common parts in the ANF representation of the output bit polynomials (which can be calculated only once). The improvement factor (compared to exhaustive search) of this direct attack is therefore a very modest $(\frac{150+7\cdot 62}{10500} + 2^{-6})^{-1} \approx 14$. The memory complexity is about $64 \cdot 2^{32} = 2^{38}$ bits and can be further reduced by iterative calculation of the output bits. We also found another interesting set of parameters: $N = \{5, 156, 214, 221, 249\}$ gives a slightly worse improvement factor of 13.

4 Improving the Direct Attack by Using Pseudo Preimages

While the direct attack seems to be worse than Fuhr's attack [2] , it can be the basis for substantial improvements. In this section we consider the generalized problem of finding a pseudo preimage, defined as a pair of a message block and chaining value $\bar{M}_i, \bar{h}_{i-1}$ such that $\mathcal{F}(\bar{M}_i, \bar{h}_{i-1}) = h_i^*$ for a given value h_i^*. Whereas in the direct attack described in the previous section we could only select our variables from the message, here we have the extra power of choosing our variables also from the chaining value bits which are mixed less extensively than

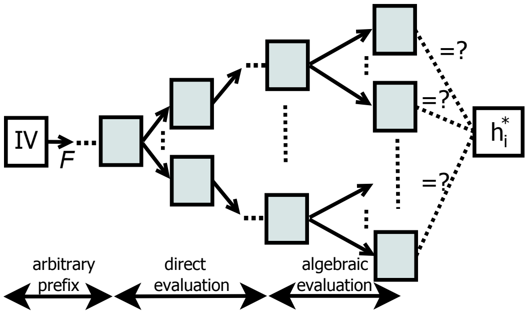

Fig. 1. A sketch of the second stage of the attack. After generating a prefix of chaining values using arbitrary message blocks, we start to traverse a tree-like structure of chaining values (shown as lightly filled boxes) using DFS: Each node is expanded into 2^{32} successor nodes by selecting the next value for the 32-bit message block in all possible ways .The tree has 8 levels so that the final level contains 2^{256} chaining values (which is roughly the number of chaining values that we need to generate in order to match the 256-bit target with high probability). The $7-th$ level nodes are not expanded by applying the Hamsi-256 compression function. Instead, we first efficiently evaluate only a small set of bits for all the 2^{32} possible message blocks. We then execute the compression function only for the messages for which the evaluation of those bits match the corresponding values of the target h_i^*. The attack succeeds once we find an $8-th$ level node that matches the target.

the message bits by the Hamsi-256 compression function. By carefully choosing these variables, we can lower the degree of the polynomials, allowing us to compute these outputs more efficiently compared to the direct attack.

Our improved attack exploits the very interesting observations made by Thomas Fuhr in section 3 of [2]. For a given message block M, we select our variables from the state that precedes the first Sbox layer as follows: Let $x^{(j)}$ denote the $j'th$ bit of the 32-bit word x. We define one variable bit $x^{(j)} \in \{0,1\}$ for each j such that $s_{14}^{(j)} = 1$ and set $s_2^{(j)} = x^{(j)}$, $s_{10}^{(j)} = \overline{x^{(j)}}$. In addition, we define one variable bit $y^{(j)} \in \{0,1\}$ for each j such that $s_1^{(j)} = 1$, $s_9^{(j)} = 0$ and set $s_5^{(j)} = y^{(j)}$, $s_{13}^{(j)} = \overline{y^{(j)}}$. According to [2], after 2 rounds of the compression function of Hamsi-256, the state bits depend linearly on our variable set: We chose our variable set such that after the first Sbox layer, only s_2, s_{13} depend linearly on our variable set. After the first round, only s_2, s_7, s_8, s_{13} depend linearly on our variable set. After the second Sbox layer, the dependency of the state on the variables remains linear and the second diffusion layer does not change this property. Note that for a random message, we expect $|V_1| = |\{x^{(j)}\}_{j \in j_x}| = 16$, $|V_2| = |\{y^{(j)}\}_{j \in j_y}| = 8$. We define $V = V_1 \cup V_2$ and for a random message we expect $|V| = 24$.

The observations of [2] allow us to select a relatively large set of variables such that the degree of all the output bits in those variables is 3. In addition, there are 28 specific output bits that depend only on 16 state bits after 2 rounds of the compression function. The indexes of these bits are $150-156, 182-188, 214-220, 246-252$. Moreover, each one of these output bits usually does not depend on all of our input variables. We calculate for each of these output bits the variables on which it actually depends, and efficiently enumerate (over all their possible values) only a certain subset of output polynomials (whose size we denote by α), called the *analyzed polynomials* or *analyzed bits*. We note that the dependencies of the 28 output bits on our variable set are influenced by the values of the message and the chaining value, but there are certain patterns that are common to most messages and chaining value pairs. For example, if our variable set contains 21 variables, there is usually an output bit which depends on only 12 of our variables, another 1 or 2 output bits that depend on 13 of our variables, another 2 or 3 output bits that depend on 14 of our variables and so forth.

4.1 The Polynomial Analysis Algorithm

All the 28 polynomials defined above have degree of at most 3 in the input variables. Given a message, a corresponding set of variables and a chaining value, we first interpolate the linear state bit polynomials of Hamsi-256 after 2 rounds. We can then use the optimized Moebius transform of section 3.2 (adapted to cubic polynomials) to efficiently evaluate any cubic polynomial, which corresponds to output bits $150-156, 182-188, 214-220, 246-252$ over all its inputs. These values are written into an array of size $2^{|S|}$, where $S \subseteq V$ is the set of input variables on which this polynomial depends. The enumeration algorithm starts with an initialization phase that interpolates the coefficients of the cubic polynomial by evaluating the function $\sum_{i=0}^{3} \binom{|S|}{i}$ times and performing $\sum_{i=0}^{3} 2^i \binom{|S|}{i}$ bit operations. The evaluation can be done by running the compression function, but we can use the second round polynomials in order to speed up this process: Given that we know the values of the 16 polynomials that are input to the third round, we can calculate the value of the output bit by evaluating 4 Sboxes and summing 4 of their outputs. An evaluation of one Hamsi Sbox output bit requires 8 bit operations (computing the 4 Sbox outputs requires 14 bit operations, but this number is reduced for individual bits), and the sum requires 3 more bit operations giving a total of 35 bit operations per evaluation. The 16 linear polynomials can be efficiently evaluated using a simple differential method which requires an average of 16 bit operations per evaluation. In total, one evaluation requires $16+35=51$ bit operations and the initialization step of the enumeration algorithm requires $\sum_{i=0}^{3} (51+2^i) \binom{|S|}{i}$ bit operations. After optimizations similar to the ones performed in section 3.2 (which exploit the sparseness of the coefficients in the array in most iterations of the algorithm), we get that the algorithm itself requires an additional number of $4 \cdot 2^{|S|}$ bit operations.

4.2 The Query Algorithm

Assume that we have already analyzed polynomials p_i for $1 \leq i \leq \alpha$ where p_i depends on a subset S_i of the variables. The output of the enumeration of each p_i is a table of size $2^{|S_i|}$. These α tables define a set of about $2^{|V|-\alpha}$ possible values for the variables such that when they are used as chaining value bits which are plugged into the compression function, the values of the α analyzed output bits match those of the target. Clearly, the remaining values of the variables that do not match the target can be safely discarded. However, these $2^{|V|-\alpha}$ solutions are only implicitly given by the tables and we have to efficiently obtain their explicit representation.

For example, assume that our set of variables is $V = \{v_1, v_2, v_3, v_4\}$, and we have analyzed polynomials p_1 that depends on $S_1 = \{v_1, v_2, v_4\}$, and p_2 that depends on $S_2 = \{v_2, v_3, v_4\}$. The table obtained after analyzing p_1 contains $2^3 = 8$ entries (an entry for each possible value of the variables of S_1). Out of these 8 entries, only entries $000, 010, 110, 111$ have a value that matches the value of the corresponding bit of the target. The other 4 entries have the complementary value (which does not match the value of the corresponding bit of the target). Note that each entry actually corresponds to two assignments of the 4 variables (For example, the point 001 corresponds to the assignments 0001 and 0011). Out of the 8 entries of the table obtained after analyzing p_2, the entries $000, 011, 110$ have a value that matches the value of the corresponding bit of the target. Our goal is to find the assignments to the 4 variables whose corresponding entries in both tables match the bits of the target. The explicit set of solutions in our example contains the points $0000, 0110, 1110$.

A naive approach in order to obtain an explicit representation of the solutions is to iterate the possible $2^{|V|}$ values for the variables, and check whether the value of the entry that corresponds to the value of the variables in each of one of the tables matches the value of the corresponding analyzed bit (note that we can discard a potential solution once the entry value in one of the tables does not match the value of the corresponding analyzed bit). This algorithm requires at least $2^{|V|}$ bit operations. We can easily save a factor of 2 by iterating only the values that match the target in one of the tables. However, we can do even better by considering the actual variable sets on which each analyzed output bit depends. The details and analysis of the improved query algorithm are specified in appendix C. Its expected complexity is $|S_1| + 2^{|S_1 \bigcup S_2|-1} + ... + 2^{|\bigcup_{i=1}^{\alpha} S_i|-\alpha+1}$ bit operations. Note that this complexity estimate is not symmetric with respect to the various S_i's, and thus different orders of analyzing the various tables will yield different complexities.

4.3 Post Filtering the Solutions

After the query algorithm, we are left with $2^{|V|-\alpha}$ solutions and we have to determine whether they indeed give a preimage which matches all the 256 bits of the given chaining value h_i^*. One option is to simply run the compression function and check whether the solutions match the target. However, it is more efficient to apply the following post filtering algorithm first.

- For each solution, evaluate the remaining $28 - \alpha$ output bits (that were not analyzed) one by one, and compare the output to the corresponding value of h_i^*. If the value of an output bit does not match the value of the corresponding target bit, drop the solution.

For each solution, we expect to evaluate 2 additional bits (we always evaluate one additional bit, a second bit is evaluated with probability 0.5, a third with probability 0.25, and so forth). Evaluating a bit requires evaluation of the 16 input linear polynomials up to round 2 plus 35 additional bit operations for the Sbox and XOR evaluations. A random linear polynomial in the $|V|$ input bits has about $\frac{|V|}{2}$ non zero coefficients, but this is not the case here. Our special choice of variables makes them diffuse slowly into the state of Hamsi-256, and as a result, our linear polynomials are very sparse and require about 3 bit operations per evaluation. The 2 evaluations thus require $2(35 + 3 \cdot 16) = 166$ bit operations. The post filtering requires in total about $166 \cdot 2^{|V|-\alpha}$ bit operations. The number of solutions that remain after the post filtering is about $2^{|V|-28}$ (i.e. we expect to have less than one solution per system on average if $|V| < 28$), and running the compression function after the post filtering requires negligible time.

4.4 Finding a Good Sequence of Analyzed Bits

In the previous sections we designed and calculated the complexities of the polynomial analysis algorithm, the query algorithm, and the post filtering algorithm. Given the sets $S_1, ..., S_{28}$ that correspond to the potential analyzed bits, we would like to find a good sequence of analyzed bits (of size α) which minimizes the complexity of the attack. Since there are many possible sequences, exhaustive search for the optimal sequence of analyzed bits is too expensive and thus we used a heuristic algorithm for this problem. A naive greedy algorithm which iteratively builds the sequence by selecting the next analyzed bit i that minimizes the added complexity $51|S|^3 + 3 \cdot 2^{|S|} + 2^{|\bigcup_{j=1}^{i} S_j|-i+1}$ seems to give reasonable results, but we got even better results by combining the greedy algorithm with exhaustive search over short sequences, as described next.

1. Given the dependencies of the 28 potential analyzed bits, exhaustively search for the optimal sequence of 3 analyzed bits that minimizes the sum of complexities of the query algorithm and their analysis.
2. Fill in the remaining $28-3 = 25$ bits of the sequence by iteratively searching for the next analyzed bit i that minimizes the added complexity $51|S|^3 + 3 \cdot 2^{|S|} + 2^{|\bigcup_{j=1}^{i} S_j|-i+1}$ (the post-filtering complexity is the same given the value of i).
3. Determine the length of the sequence α by calculating the total complexity of the attack for each possible value of $1 \leq \alpha \leq 28$ and truncate the sequence of 28 bits to size α.

The first step involves exhaustive search over $\frac{28!}{25!} < 2^{14.5}$ sequences, each requires a union of sets of at most $|V|$ variables represented as bit arrays, and an addition operation. The union requires $|V|$ bit operations and the addition a few more bit operations since the terms $51|S|^3 + 3 \cdot 2^{|S|}$ are computed only once and can be rounded in order to nullify the least significant bits. Assuming that $|V| < 2^5$, the complexity of the first step is about $2^{19.5}$ bit operations, which can be easily reduced to about $2^{18.5}$ by considering the sequences in a more clever way. The second and third steps take negligible time compared to the first step. Note that this algorithm is performed before analyzing the polynomials, although it is specified last.

4.5 Details of the Pseudo Preimage Attack on Hamsi-256

The details and analysis of the pseudo preimage attack on Hamsi-256 are specified below. The input of the algorithm is a chaining value h_i^*, and its output is a message block $\bar{M}_i$ and a chaining value $\bar{h}_{i-1}$ such that $\mathcal{F}(\bar{M}_i, \bar{h}_{i-1}) = h_i^*$.

1. Generate the next message block $\bar{M}_i$ (starting from the zero block, and incrementing its value each time this step is performed).
2. Compute the set of variables $V' = V_1 \cup V_2$ according to $\bar{M}_i$. If $|V'| < 21$ then discard the message and go to step 1. Otherwise, obtain the final set of 21 variables V for the current message block by dropping $|V'| - 21$ variables from V'. The variables that are dropped are arbitrarily chosen from the set V_1 (the variables are dropped from V_1 since the variables of V_2 tend to diffuse more slowly into the state of Hamsi-256, as noted in section 4 of [2]).
3. Generate the next partial chaining value $\bar{h}_{i-1}$, which does not assign values to the variables (starting from the zero partial chaining value each time step 1 is performed, and incrementing its value each time this step is performed). If no more partial chaining values exist, go to 1.
4. Given $\bar{M}_i$, V and $\bar{h}_{i-1}$, interpolate the linear state bit polynomials of Hamsi-256 after 2 rounds.
5. For each of the 28 output bits $(150 - 156, 182 - 188, 214 - 220, 246 - 252)$, determine the variable subset on which it depends. This is done by retrieving the 16 linear second round state bits on which the output bit depends, and then performing a union over the variable subsets on which the 16 state bits depend.
6. Determine the heuristically best sequence of analyzed bits according to the algorithm in section 4.4.
7. Analyze the selected polynomials according to section 4.1.
8. Use the query algorithm of section 4.2 to determine the set of solutions.
9. Post filter the solutions according to section 4.3. If no solutions remain, go to step 3.
10. For each remaining solution, compute the compression function after assigning the value of the solution to the unspecified part of the partial chaining value, and check whether the output is equal to the target. If there is a solution for which equality holds, return the message and full chaining value. Otherwise, go to step 3.

In order to find at least one pseudo preimage with high probability, we must verify that we do not use too many degrees of freedom after throwing away messages and allocating the variables. We start with 32 degrees of freedom since the input to the Hamsi-256 compression function contains $256 + 32 = 288$ bits, (32 message bits and 256 chaining value bits) and the output of the compression function contains only 256 bits. We lose less than 0.5 degrees of freedom by throwing away messages for which the number of variables is too small. In addition, every variable sets one constraint on the input of the compression function and reduces the number of possible inputs to the compression function by a factor of 2. Thus, we lose a degree of freedom per allocated variable and less than 21.5 degrees of freedom overall. In total, we remain with a bit more than $32 - 21.5 = 10.5$ degrees of freedom which are expected to result in more than 2^{10} pseudo preimages for a random target.

We now estimate the complexity of the pseudo preimage attack: The complexity of some steps can be easily computed: For a given set of variables, step 4 of the algorithms requires 22 compression function evaluations of Hamsi-256 and $21 \cdot 512 < 2^{14}$ bit operations. Step 5 takes negligible time. Step 6 requires about $2^{18.5}$ bit operations. Step 10 requires $2^{|V|-28}$ compression function evaluations, which takes negligible time compared to the other steps of the attack. However, the complexity of the main steps of the attack $7 - 9$ cannot be easily computed since it depends on the message and the value of the chaining value used. Thus, we can only estimate the complexity of the attack by running simulations for randomly chosen messages and chaining values. In each simulation, we estimate the complexity of the attack by summing the complexities of the steps above with the complexity of steps $7 - 9$, as calculated in section 4.4. After thousands of simulations we found that for about 95% of messages and chaining values the attack is faster than exhaustive search by a factor which is at least 2^{13}. The average complexity of the attack is slightly better than $2^{256-13.5} = 2^{242.5}$ compression function evaluations.

Interestingly, the techniques of our pseudo preimage attack can also be used to speed up generic pseudo collision search algorithms on Hamsi-256 that are based on cycle detection algorithms (such as Floyd's algorithm [11]). The details of the pseudo collision attack are described in appendix D.

5 Using Pseudo Preimages to Obtain Second Preimages for Hamsi-256

Given a message $M = M_1^*||M_2^*||...||M_\ell^*$ with $\ell \geq 9$, we can use the naive meet-in-the-middle algorithm described in appendix B in order to find an expected number of $2^{13.5/2} = 2^{6.75}$ pseudo preimages and use them as targets for the second preimage attack. This gives a total complexity of about $2^{256-5.75} = 2^{250.25}$ compression function evaluations. However, we can do better by using the result of section 3: Recall that our algorithm for finding pseudo preimages has more than 10 degrees of freedom left. We use 5 of the remaining degrees of freedom to set the input bits that correspond to the output bits of the set

$N = \{5, 156, 214, 221, 249\}$ in all the pseudo preimages to some fixed value. As specified in section 3.3, the set N represents the target bits for the direct second preimage attack on Hamsi-256 and this choice allows us to speed up the second phase by a factor of about $13 \approx 2^{3.7}$. In the first phase of the attack, the bits of N actually function as input bits to the pseudo preimage search algorithm. The details of the algorithm are specified below, where x is a numeric parameter:

1. Choose a target block with index of at least 9 (i.e. h_i^* with $i \geq 9$) and use the pseudo preimage search algorithm to find 2^x pseudo preimages in which the set of input bits $\{5, 156+128, 214+128, 221+128, 249+128\}$ is fixed to an arbitrary value. Note that the number 128 is added to some indexes of N due to the truncation of the output of the compression function.
2. Use the direct second preimage search algorithm to find a second preimage to one of the 2^x pseudo preimages found in the previous step.

We note that we still have $10 - 5 = 5$ degrees of freedom left, so we must choose $x \leq 5$ in the first step. The complexity of step 1 is about $2^{256-13.5+x} = 2^{242.5+x}$ compression function evaluations. The complexity of step 2 is about $2^{256-x-3.7}$ compression function evaluations. To optimize the attack, we choose $2^x = 30$, i.e $x \approx 4.9$ for which the total complexity of the attack is about $2^{248.4}$, which is about $2^{7.6} \approx 200$ times better than exhaustive search.

The algorithm presented above works for any message that contains at least 9 blocks. However, this restriction can be removed with little additional cost using an observation made by an anonymous referee of this paper: Since the 64-bit message length is encoded in the last two 32-bit blocks, we can find a pseudo preimage of the last intermediate chaining value with a non-zero message block. The 32 bits of the message block function as the most significant bits of the message length of our second preimage, which now contains enough blocks for the attack. The chaining value of the pseudo preimage gives us the target which we require for the algorithm above.

In addition, it is possible to improve the algorithm further by building a *layered hash tree*, similar to the one used in [10]. The optimized algorithm yields a less marginal improvement factor of $2^9 = 512$ over exhaustive search, which is about 20 times better than the attack published by Thomas Fuhr [2]. The details of this algorithm are specified in appendix E

6 Second Preimages for Longer Messages of Hamsi-256

The best known generic algorithm for finding second preimages for any Merkle-Damgård construction of hash functions is due to Kelsey and Schneier [3]. The algorithm needs to undergo a slight adaptation in order to be applied to the special structure of Hamsi-256 (see [2]). The complexity of the generic algorithm for Hamsi-256 is $k \cdot 2^{128} + 2^{256-k}$, where the message length satisfies $\ell \geq 4k + 2^k + 8$. Hence, the algorithm developed in the previous section is better than the generic algorithm only for $k \leq 9$, i.e. for messages that contain at most $4 \cdot 9 + 2^9 + 8 = 556$ blocks. For longer messages, we design a different algorithm that combines the techniques used in section 3 with a modified version of the Kelsey and Schneier algorithm. We elaborate only on the parts of the Kelsey and Schneier algorithm that are relevant to our modified attack.

Given an ℓ block message, in the the first phase of the Kelsey and Schneier algorithm, the attacker generates a (p, q) expandable message for $p = 4k$ and $q = 4k + 2^k - 1$ such that $q + 8 \leq \ell - 1$. This phase is left unchanged. We concentrate on the second phase of the Kelsey and Schneier algorithm, where we apply the compression function from a common chaining value and try to connect to one of the chaining values obtained by one of the invocations of the compression function during the computation of the hash of the given message. If the message is of size about 2^k, the complexity of this phase is 2^{256-k} compression function evaluations, which forms the bottleneck of the attack (assuming $k < 128$). Similarly to section 3, the idea is to speed up this phase simply by efficiently computing several bits of the output for all possible 2^{32} messages and filtering out messages which do not connect to any of the targets. Assuming that we efficiently compute the values of x output bits, then we still need to run the compression function a factor of 2^{-x+k} times for $x > k$ compared to the original algorithm.

Unlike section 3, a significant portion of the work here involves computing the output bits (almost) directly, and a smaller portion of the work involves analysis of the second round bits. The output bits are of degree 18 which is too high to be analyzed efficiently. However, we can exploit polynomials of a lower degree relatively easily. As in section 3.3, we use the ANF form of the output bits as a function of the second round bits. The symbolic representation is of degree 3 and we would like to get equations of degree 2. We remove all terms of degree lower than 3 in the ANF form. We then linearize the system of polynomials by assigning each distinct term of degree 3 a dedicated variable. We perform Gaussian Elimination on the linearized system and get a system in which about 120 rows contain only 1 variable and the rest of the rows contain 2 variables (each linearized variable represents 3 variables of round 2 multiplied together). This is of course not sufficient in order to reduce the degree. However, these linearized simple expressions (composed of 3 variables of degree 6 in the message bits) can be handled separately by the technique specified in section 3.2. We select a set of x linear combinations from the rows which contain only one linearized variables. The rest of the polynomial is of degree $2 \cdot 3 \cdot 2 = 12$ in the message bits, and is analyzed slightly differently. The analysis algorithm for such a linear combination is specified below. Its input is an arbitrary chaining value h and it outputs an array of size 2^{32} that contains the evaluations of the linear combination of the output bits for all possible 2^{32} message blocks.

1. Analyze the 3 second round variables that appear in the expression of the linearized variable of the linear combination, as specified in section 3.2.
2. Evaluate the remainder of the output bit combination on all input vectors of hamming weight ≤ 12 and store the results.
3. Interpolate the coefficients of the output bit combination: Place all its values in an array of size 2^{32}, where the values of entries of hamming weight ≥ 13

are set to zero. Then apply the Moebius transform [8] on the array and take only the coefficients of hamming weight ≤ 12 (the rest are known to be 0).

4. Apply the Moebius transform once more on the array and obtain the evaluations of the polynomial (not including the linearized variable) for all 2^{32} possible input values.
5. Add the values of the linearized variable to the array by computing it from the arrays produced in step 1.

Step 1 requires $3 \cdot 7 \cdot 2^{32} = 21 \cdot 2^{32}$ bit operations. Step 2 requires $\sum_{i=0}^{12} \binom{32}{i} \approx \frac{2^{32}}{10}$ compression function evaluations (which need to be performed once per chaining value). In addition, step 2 requires several bit operations to compute the value of the linear combination. Most of the linear combinations contain fewer than 40 additions and step 2 requires additional $40 \cdot 0.1 \cdot 2^{32} = 4 \cdot 2^{32}$ bit operations. Step 3 requires an application of the Moebius transform, which takes $16 \cdot 2^{32}$ bit operations. However, only about 0.1 of the entries of the array are relevant (the others are not accessed), hence the complexity is less than $2 \cdot 2^{32}$ bit operations. Step 4 requires the full $16 \cdot 2^{32}$ bit operations. Step 5 requires additional $3 \cdot 2^{32}$ bit operations. In total, the algorithm requires about $(21+4+2+16+3) \cdot 2^{32} = 46 \cdot 2^{32}$ bit operations in addition to the $\frac{2^{32}}{10}$ compression function evaluations that are performed globally.

The algorithm to find the second preimage is specified below, where x is a numeric parameter. It get as an input a message $M_1^*||M_2^*||...||M_\ell^*$ and outputs a message of the same length with the same Hamsi-256 hash value.

1. Generate a p, q expandable message for $p = 4k$ and $q = 4k + 2^k - 1$ such that $q + 8 \leq \ell - 1$.
2. Choose a set of x output bit combinations from the Gaussian elimination of the third round output bits in terms of the second round variables, such that each of these combinations contains a single expression of 3 second round bits multiplied together.
3. Compute and store all the values of the x output bit combinations of all the target chaining values h_i^* for $p + 8 \leq i \leq q + 8$.
4. Choose the common digest value of the expandable message h as a chaining variable and traverse the chaining value tree rooted at h using DFS by generating the next value for message blocks $M_1, M_2, ..., M_7$ (as shown in figure 1).
5. Compute the next chaining value $h_7 = \mathcal{F}(M_1, M_2, ..., M_7, h)$.
6. Analyze each one of the x output bit combinations as specified above for all possible 2^{32} values for the message block M_8, with the input chaining value h_7.
7. Traverse the x bit arrays and check whether the values of the output combinations match the values of the combinations of the target chaining values h_i^* for $p + 8 \leq i \leq q + 8$, for each possible value of the message block M_8. Store all the messages for which there is a match.

8. For each message block M_8 stored in the previous step, evaluate the full compression function and check whether $\mathcal{F}(M_8, h_7) = h_i^*$ for $p+8 \leq i \leq q+8$. If equality holds, output the message $\mu_{i-8}||M_1||M_2||...||M_8||M^*_{i+1}||...||M^*_\ell$, where μ_{i-8} is a message prefix of size $i-8$ blocks (computed from the expandable message) such that $h = \mathcal{F}(\mu_{i-8}, IV)$. Otherwise, if there is no match, go to step 4.

We analyze the complexity of the algorithm per chaining value h_7 (i.e. steps $6-8$) in order to calculate the improvement factor of the attack over the generic algorithm. The Kelsey and Schneier algorithm requires 2^{32} compression function evaluations per chaining value, whereas we use only $\frac{2^{32}}{10}$ compression function evaluations. In addition, we require $46 \cdot x \cdot 2^{32}$ bit operations in step 6. However, we can optimize the complexity of this step for a group of combinations by taking combinations in which the linearized expressions share some common variables of the second round (which need to be analyzed only once). In particular, we can easily select a group of x combinations in which the x linearized expressions depend only on $2 \cdot x$ (instead of $3 \cdot x$) variables of the second round. This reduces the number of bit operations in step 4 to $39 \cdot x \cdot 2^{32}$. The improvement factor of the attack is thus $(\frac{1}{10} + \frac{39x}{10500} + 2^{-x+k})^{-1}$. By selecting an optimal value for x, we get a total improvement factor which is between 6 and 4 for all messages of practical length containing up to 2^{30} 32-bit blocks, whereas Fuhr's attacks [2] becomes worse than the generic attack for all messages which are longer than 96 blocks.

7 Conclusions

In this paper, we presented several second preimage attacks on Hamsi-256 that are based on polynomial enumeration algorithms. Our attacks are faster than Fuhr's attack for all message lengths, and unlike Fuhr's attack they are faster than the generic Kelsey and Schneier attack for all practical message sizes. Our new techniques can be applied in principle to other hash algorithms whose compression function can be described by a low degree multivariate polynomial, and demonstrate the potential vulnerability of such schemes to advanced algebraic attacks. In addition, our techniques can be used to speed up exhaustive search on secret key algorithms (such as block cipher, stream ciphers and MACs) that can be described by a low degree multivariate polynomial in the key bits. However, hash function designs with a stronger finalization function and an intermediate state that is bigger than the output (i.e "wide-pipe" designs), seem to better resist our attack.

Acknowledgements: The authors thank Orr Dunkelman and Nathan Keller for helpful discussions that led to this paper. The authors also thank the anonymous referees for their very helpful comments on this paper.

References

1. Özgül Küçük. The hash function hamsi. Submission to NIST (updated), 2009.

2. Thomas Fuhr. Finding Second Preimages of Short Messages for Hamsi-256. In *ASIACRYPT*, 2010.
3. John Kelsey and Bruce Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In *EUROCRYPT*, pages 474-490, 2005.
4. Jean-Philippe Aumasson and Emilia Käsper and Lars Ramkilde Knudsen and Krystian Matusiewicz and Rune Ødegård and Thomas Peyrin and Martin Schläffer. Distinguishers for the Compression Function and Output Transformation of Hamsi-256. In *ACISP*, 2010.
5. Christina Boura and Anne Canteaut. Zero-sum Distinguishers for Iterated Permutations and Application to Keccak-f and Hamsi-256 . In *SAC* 2010.
6. Çagdas Çalik and Meltem Sönmez Turan. Message Recovery and Pseudo-preimage Attacks on the Compression Function of Hamsi-256. In *LATINCRYPT*, pages 205-221, 2010.
7. Meiqin Wang, Xiaoyun Wang, Keting Jia, Wei Wang. New Pseudo-Near-Collision Attack on Reduced-Round of Hamsi-256. Cryptology ePrint Archive, Report 2009/484, 2009.
8. Antoine Joux. Algorithmic Cryptanalysis. Chapman & Hall, page 285-286.
9. Charles Bouillaguet and Hsieh-Chung Chen and Chen-Mou Cheng and Tony Chou and Ruben Niederhagen and Adi Shamir and Bo-Yin Yang. Fast Exhaustive Search for Polynomial Systems in F_2. In *CHES*, 2010.
10. Gaëtan Leurent. MD4 is Not One-Way. In *FSE*, pages 412-428, 2008.
11. Antoine Joux. Algorithmic Cryptanalysis. Chapman & Hall, page 225-226.

A Appendix: Parameters For the Direct Attack on Hamsi-256

The 56 second round bits on which the set $N = \{5, 156, 184, 214, 221, 249\}$ depends are listed below:
{3, 9, 18, 28, 44, 56, 63, 68, 79, 86, 91, 92, 93, 96, 107, 121, 131, 156, 184, 191, 196, 214, 219, 220, 221, 224, 249, 256, 259, 265, 274, 275, 284, 300, 312, 319, 324, 335, 342, 347, 348, 349, 352, 363, 377, 387, 412, 440, 447, 452, 470, 475, 476, 477, 480, 505}. The 6 simple sums for the sets $N = \{5, 156, 184, 214, 221, 249\}$ are given in the table below:

Table 2. The 6 simple sums of the second round variables denoted by x_i for $0 \leq i < 512$ for the output bits $N = \{5, 156, 184, 214, 221, 249\}$.

Output Bit	Simple Sum
5	$x_9 + x_{18} + x_{19} + x_{63} + x_{86} + x_{92} + x_{121} + x_{137} + x_{146} + x_{147}$ $+x_{214} + x_{249} + x_{265} + x_{274} + x_{275} + x_{319} + x_{342} + x_{393} + x_{402}$ $+x_{403} + x_{470} + x_{476} + x_{505}$
156	$x_3 + x_{28} + x_{79} + x_{156} + x_{191} + x_{207} + x_{259} + x_{284} + x_{387} + x_{412}$ $+x_{447} + x_{463}$
184	$1 + x_{56} + x_{63} + x_{107} + x_{235} + x_{319} + x_{347} + x_{447} + x_{475} + x_{491}$
214	$1 + x_9 + x_{86} + x_{121} + x_{137} + x_{v214} + x_{249} + x_{265} + x_{342} + x_{393}$ $+x_{470} + x_{477} + x_{505}$
221	$1 + x_{18} + x_{63} + x_{68} + x_{92} + x_{96} + x_{146} + x_{224} + x_{274} + x_{319}$ $+x_{324} + x_{352} + x_{402} + x_{452} + x_{476} + x_{477}$
249	$1 + x_{28} + x_{44} + x_{96} + x_{121} + x_{156} + x_{172} + x_{224} + x_{249} + x_{300}$ $+x_{377} + x_{412} + x_{428} + x_{480} + x_{505}$

B Appendix: Finding a Second Preimage of Hamsi-256 Using a Pseudo Preimage Search Algorithm

We describe the well-known meet-in-the-middle algorithm for finding a second preimage of Hamsi-256 using a pseudo preimage search algorithm that runs in time 2^{256-y} compression function evaluations (for some $y > 0$). Assume that we are given a message $M = M_1^*||M_2^*||...||M_\ell^*$ with $\ell \geq 9$ and let x be a numeric parameter which will be optimized later. The algorithm (adapted to the special structure of Hamsi-256) is described below.

1. Use the pseudo preimage search algorithm to find 2^x pseudo preimages of h_i^* for any fixed $i \geq 9$ (i.e. a chaining value obtained after at least 9 compression function invocations on the given message.
2. Generate arbitrary $i-9$ message blocks $M_1, M_2..., M_{i-9}$ and compute $h_{i-9} = \mathcal{F}(M_1, M_2, ..., M_{i-9}, IV)$. Generate 8-block messages $M_{i-8}, M_{i-7}, ..., M_{i-1}$ and check whether $\mathcal{F}(M_{i-8}, M_{i-7}, ..., M_{i-1}, h_{i-9})$ matches one of the chaining values of the pseudo preimages found in the first step. Assume that $\mathcal{F}(M_{i-8}, M_{i-7}, ..., M_{i-1}, h_{i-9}) = \bar{h}_{i-1}$, where $\bar{h}_{i-1}, \bar{M}_i$ is a pseudo preimage found in the first step.
   Then, the message $M_1||M_2||...||M_{i-9}||M_{i-8}||...||M_{i-1}||\bar{M}_i||M_{i+1}^*||...||M_\ell^*$ is a second preimage of M.

The complexity of step 1 of the algorithms is $2^{256-y+x}$ compression function evaluations. The complexity of step 2 is 2^{256-x} compression function evaluations. To optimize the complexity of the attack, we need that both steps run in equal time. Thus, we choose $x = \frac{y}{2}$ for which the total complexity of the attack is $2^{256-\frac{y}{2}+1}$.

C Appendix: The Detailed Query Algorithm

The basic data structure that we use is a table T that assigns a bit value to all the points in a subset S of our variable set V. The table contains $2^{|S|}$ bit entries. An entry with a 0 value represents a point is the subset, or an assignment to the subset of the variables, which is not in our solution set. When expanded to all the $|V|$ variables, this single point disqualifies $2^{|V|-|S|}$ points from our solution set. Note that all our α enumeration tables are of this type, except that in some tables an entry with a 1 value represents a disqualified point in the subset (this occurs if the value of the corresponding analyzed bit in the target is 0). However, this is just a minor representation issue.

The main procedure of the query algorithm involves a table T_i that assigns values to all the points in a subset S_i and a query object q that assigns values to a subset of the variables $R \subseteq V$.

1. Expand the query object to $2^{|S_i \setminus R|}$ queries that assign all possible values to the set $|S_i \setminus R|$ and the same values as assigned by q to the set R (Note that each query assigns values to the set $S_i \bigcup R$).

2. For each query, check whether the table disqualifies it: If it does, discard it. Otherwise, if t_i is the last table (i.e. $i = \alpha$), return the query as a solution, otherwise ($i < \alpha$), pass it to the next table T_{i+1}.

The query algorithm basically calls the procedure above with T_1 and a query for the empty subset.

We estimate the complexity of the query algorithm by the total number of queries that are generated during its execution. In this analysis, we assume that half of the queries are discarded by each table and half are passed to the next table (or returned as solutions by the last table). The expected number of queries is then $|S_1| + 2^{|S_1 \cup S_2|-1} + ... + 2^{|\cup_{i=1}^{\alpha} S_i|-\alpha+1}$.

D Appendix: The Pseudo Collision Attack on Hamsi-256

We show how to use the techniques of our pseudo preimage attack (described in section 4) to speed up generic pseudo collision search algorithms on Hamsi-256 that are based on cycle detection algorithms. Let $v \leq 28$ be a numeric parameter that will be specified later. Let R by an arbitrary subset of size v of output bits $150-156, 182-188, 214-220, 246-252$ (as described in section 4, these are output bits that depend only on 16 state bits after 2 rounds of the compression function). We select an arbitrary fixed value for the bits of R which will function as a static target for our fast polynomial enumeration techniques and reduce the effective output size for finding a collision to $256-v$. Given a 32-bit message block $\bar{M}$, denote by $\bar{M}'$ the first v bits of $\bar{M}$ and by $\bar{M}''$ its remaining $32-v$ bits. We now describe the iteration function that is used during the cycle detection algorithm. The input and output of the iteration function are of size $256-v$ bits.

1. Set the bits of $\bar{M}'$ to the value of the first v input bits.
2. Iterate through the possible values of $\bar{M}''$ (reset to 0 on step 1, and increment each time this step is performed).
3. Given $\bar{M}$, compute the set of variables $V' = V_1 \cup V_2$ (as described in section 4). If $|V'| < v$ go to step 2. Otherwise, obtain the set V of v variables for the message block by dropping $|V'|-v$ variables from V'. The variables that are dropped are arbitrarily chosen from the set V_1.
4. Let h' be a partial chaining value in which the $2v$ bits which are assigned values that depend on V are unspecified (there are two bits that depend on each variable). Set the value of h' to be the remaining $256-2v$ bits of the input (that were not used to set $\bar{M}'$).
5. In the remaining steps, find an assignment to the variables of V that will match the target bits of R: Given $\bar{M}$, V and h', interpolate the linear state bit polynomials of Hamsi-256 after 2 rounds.
6. For each of the output bits of R, determine the variable subset on which it depends (as done in the Pseudo Preimage Attack).
7. Determine the heuristically best sequence of analyzed bits according to the algorithm in section 4.4.

8. Analyze the selected polynomials according to section 4.1.
9. Use the query algorithm of section 4.2 to determine the set of solutions. If there is no solution, go to step 2.
10. Post filter the solutions according to section 4.3. If no solutions remain, go to step 2.
11. Choose the solution with the smallest lexicographic value (or using any other deterministic algorithm), and compute the compression function after assigning the value of the solution to the unspecified part of the partial chaining value h'. The $256 - v$ output bits of the iteration function are defined to be the $256 - v$ output bits of the compression function (not including the bits of R, which have a fixed value).

Assuming that we choose $v \leq 24$, we do not expect to return to step 2 via step 3 most of the time. In addition, for any partial chaining value h' (that does not assign values to the variables), there are 2^v possible outputs defined by all the possible assignments to the variables. Thus, we expect that the value of the v output bits of R will match our pre-defined constant value for the bits of R for at least one of the 2^v possible outputs, hence we do not expect to return to step 2 from steps 9 and 10 as well. As a result, for the complexity analysis, we assume that steps $3 - 11$ are performed once per invocation of the iteration function. However, since the iteration function is invoked numerous times during the cycle detection algorithm, it is likely that we run out of values for $\bar{M}''$ in step 2 without finding a solution more than once. In this case, we can simply increment the value of $\bar{M}'$ or h'. This may result in false alarms (in case we jump to a node that was already visited), but this happens with negligible probability.

Assuming that the time complexity of the iteration function is 2^y compression function evaluations, then the modified pseudo collision search algorithm is expected to find a pseudo collision in time $2^{\frac{256-v}{2}+y} = 2^{128-\frac{v}{2}+y}$ (since the effective output size is $256 - v$). Similarly to our pseudo preimage attack, the complexity of the attack (and the optimal value for v) needs to be determined via simulations. The best value for v that we found is 18 and our best pseudo collision attack runs in time 2^{125} which is an improvement of about $2^3 = 8$ compared to the generic algorithms for finding pseudo collisions.

E Appendix: Details of the Improved Short Message Attack on Hamsi-256

We describe the details of the improved short message second preimage attack, given a message $M_1^*||M_2^*||...||M_\ell^*$ such that $\ell > 40$. It uses the *layered hash tree* construction, as described in [10]: We start by selecting the root of the tree from the chaining values that are generated during the computation of the hash of the message. We then find a pseudo preimage of the root and add it to the tree. We continue building the tree by adding a pseudo preimage of one of the two nodes of the tree, and so forth. Assuming that we need a workload of 2^s to find a pseudo preimage of a single target value, and we have a set of k target values in our pseudo preimage tree, we can find a pseudo preimage of one of them in time $\frac{2^s}{k}$. Thus, the complexity of building a tree containing 2^x pseudo preimages is $2^s \sum_{k=1}^{2^x} \frac{1}{k}$.

Let v ,x and y be numeric parameters that will be specified later. Let R by an arbitrary subset of size y of output bits $150-156, 182-188, 214-220, 246-252$. The idea is to build a tree of preimages in which the bits of R are fixed, and will function as a static target for our fast polynomial enumeration techniques. The root is selected as an arbitrary chaining value h_i^* $(i \geq 40)$ that is generated during the computation of the hash of the message. The algorithm for adding a node to the tree is very similar to the pseudo preimage algorithm of section 4.5. The differences are that our variable set V is of size v (instead of 21), and the analyzed bits are chosen from the set R. In addition, we have to compute more bits in the post filtering algorithm (of section 4.3), since we have to eliminate possible pseudo preimages for a larger group of targets (and not just 1, as in section 4.5). Note that the number of degrees of freedom is reduced by a bit more than $v + y$ (since we throw away a small fraction of the messages), so we must select $v + y < 32$.

According to our simulations, the best parameters the attack are $v = 20$, $y = 11$, and $x = 8$. These parameters give an expected workload of $2^s = 2^{256-11.5}$ compression function evaluations, and thus the time complexity of building the tree is $2^{256-11.5} \sum_{k=1}^{2^8} \frac{1}{k} \approx 2^{256-11.5} \cdot 6 \approx 2^{256-8.9}$.

We can improve this complexity a bit since we have the freedom to choose the 11 bits of R among the output bits $150-156, 182-188, 214-220, 246-252$. Thus, we first find 4 pseudo preimages of the root according to the algorithm of section 4.5 (which has a better workload), in which 6 of these bits are fixed to some value. With high probability these pseudo preimages will share a common value for at least 11 of the output bits $150-156, 182-188, 214-220, 246-252$ (from which we select the set R). Using this improved algorithm, the complexity of building a tree with 2^8 pseudo preimages costs about $2^{256-9.5}$ compression function evaluations. It is possible to prove (or verify experimentally) that such a layered tree with 2^8 nodes has at most 20 layers with very high probability.

After the construction of the layered hash tree, we have to find a preimage of one of the nodes of the tree. Since the tree has several layers and we want to be able to connect to any node, this is done using an expandable message (similar to our long message attack that is described in section 6). The layered hash tree is rooted at h_i^* with $i \geq 40$, and it contains (at most) 20 layers with high probability. Thus our (p, q) expandable message needs to satisfy $p \leq 20$ (to connect from block with index at least $40 - 20 = 20$) and $q - p \geq 20$ (to connect to any of the possible 20 layers). The outline of the algorithm is given below.

1. Select an arbitrary chaining value h_i^* $(i \geq 40)$ that is generated during the computation of the hash of the message.

2. Construct a layered hash tree rooted at h_i^* containing 2^8 pseudo preimages, such that the bits of R in all the pseudo preimages are fixed (as specified above).
3. Use the Kelsey and Schneier [3] algorithm to generate a (p, q) expandable message for $p = 4 \cdot 5 = 20$ and $q = 4 \cdot 5 + 2^5 - 1 = 51$.
4. Select a subset $N \subset R$ of output bits of size 5. Starting from the common digest value of the expandable message, use the efficient elimination algorithm of section 3.3 (with the fixed target bits of N) to connect to one of the nodes of the pseudo preimage tree. Once a match is found, the second preimage can be easily constructed.

The algorithm presented above works for any message that contains at least 40 blocks. However, this restriction can be removed with little additional cost, as explained in section 5. Building the tree requires an expected time of $2^{256-9.5}$ compression function evaluations. Step 3 requires negligible time compared to step 2. For a single target and an arbitrary set N of size 5 (selected from output bits $150-156, 182-188, 214-220, 246-252$), the efficient elimination algorithm of step 4 has time complexity of about 2^{256-3} compression function evaluations. Thus, for 2^8 targets the complexity of step 4 is about $2^{256-3-8} = 2^{256-11}$. Overall, the complexity of the attack is about $2^{256-9.5} + 2^{256-11} \approx 2^{256-9} = 2^{247}$ compression function evaluations, which is an improvement by a factor of 512 compared to exhaustive search.