Cryptanalysis of a hash function, and the modular subset sum problem

Chris Monico

Department of Mathematics and Statistics
Texas Tech University

January 17, 2018

Abstract

Recently, Shpilrain and Sosnovski proposed a hash function based on composition of affine maps. In this paper, we show that this hash function with its proposed parameters is not weak collision resistant, for plaintexts of size at least 1.9MB. Our approach is to reduce the preimage problem to a (very) high density instance of the Random Modular Subset Sum Problem, for which we give an algorithm capable of solving instances of the resulting size. Specifically, given plaintexts of about 1.9MB, we were able to produce other plaintexts of the same size with the same hash value in about 13 hours each, on average.

1 Introduction

Loosely speaking, a *hash function* is an efficiently computable function h with variable-length input and fixed-length output. Additionally, for most cryptographic applications it is desirable that the following inverse problems are computationally infeasible [3]:

1. *preimage resistance* (*one-wayness*): Given y in the image of h, find x' for which $h(x') = y$ (this should be hard for almost all y in the image).

2. *2nd-preimage resistance* (*weak collision resistance*): given x, find $x' \neq x$ such that $h(x) = h(x')$.

3. *collision resistance* (*strong collision resistance*): Find distinct x, x' in the domain such that $h(x) = h(x')$.

Such functions have numerous applications in cryptography, to problems such as data integrity verification and authentication [3]. Among the most currently used hash functions for these purposes are those from the SHA family [4].

In 1994, Tillich and Zémor [6] proposed a hash function by fixing a pair of matrices $A, B \in \mathrm{SL}_2(\mathbb{F}_{2^n})$, defining $\pi(0) = A$, $\pi(1) = B$, and sending the bitstring $x_1x_2\ldots x_k$ to $\pi(x_1)\pi(x_2)\ldots\pi(x_k)$. One of the main motivations for their proposal was that the algebraic

properties of this function allow an algebraic description of some of the inverse problems that should be computationally infeasible. In 2011, Grassl et.al [1] subsequently showed that the particular proposed hash function is not collision resistant.

In [5], the authors propose a related cryptographic hash function, obtained by repeated composition of affine maps. Let p be a prime, and $f_1(x) = 2x + 1$ and $f_0(x) = 3x + 1$. If $b_0, b_1, \ldots, b_n \in \{0, 1\}$ is a bit-string to be hashed, they propose computing

$$rx + s = f_{b_n} \circ f_{b_{n-1}} \circ \cdots \circ f_{b_1} \circ f_{b_0}(x) \pmod{p},$$

and using the pair $(r + s, s)$ as the hash of this string. The relationship to the Tillich-Zémor proposal is afforded by the matrix description given in Section 2. They further suggest and study the results of using a prime p around 2^{256}. The goal of this paper is to show that this hash function is not *weak collision resistant* for inputs larger than about 1.9 Megabytes (MB), with $p \approx 2^{256}$. Specifically, given $p \approx 2^{256}$ and the hash value of a bit string of known length of at least 1.9 MB, we have been able to produce other bit strings of the same length with the same hash value. Our method does not even need the original bit string - a reasonable bound on its length and the hashed value will suffice.

To do this, we reduce the problem to a dense instance of the Random Modular Subset Sum Problem (RMSSP), and give a probabilistic algorithm for solving it. We also argue heuristically that the algorithm is expected to succeed as long as the originally hashed bit string had at least n zeros and n ones for some $n \geq 2^{\sqrt{2 \log_2 p}}$, and that the expected runtime is $\mathcal{O}(n^2 \log n)$, with an implied constant small enough to keep the attack practical for $p \approx 2^{256}$.

2 Reduction to RMSSP

Throughout, let p be a fixed odd prime and

$$A = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} \in \mathrm{GL}_2(\mathbb{F}_p), \quad \text{and} \quad B = \begin{pmatrix} 3 & 1 \\ 0 & 1 \end{pmatrix} \in \mathrm{GL}_2(\mathbb{F}_p).$$

Although we never need to explicitly work with matrices, they provide a convenient description of this hash function, as was observed in [5]. If we identify the affine function $cx + d$ with the matrix $\begin{pmatrix} c & d \\ 0 & 1 \end{pmatrix}$, then $f_1(cx + d) = 2cx + (2d + 1)$ which is the function identified with $A \begin{pmatrix} c & d \\ 0 & 1 \end{pmatrix}$. Similarly, left-multiplication by B corresponds to composition with f_2.

Suppose now we are given a pair $(x, y) \in \mathbb{F}_p^2$ which is the hashed value of some bit string of known length L. We first invert the final operation of adding the second coordinate to the first, to get $(r, s) = (x - y, y)$ modulo p. We are then trying to find a word in the matrices A and B which evaluates to $\begin{pmatrix} r & s \\ 0 & 1 \end{pmatrix}$. Since $\det(A) = 2$ and $\det(B) = 3$, we therefore have that $r = 2^a 3^b$, where a is the number of ones in the original bit string and b is the number of zeros in that bit string. We have immediately that $a + b = L$. Since L is known, a and b can be recovered with $\mathcal{O}(L \log L)$ operations over $\mathbb{F}_p$; for example, by precomputing and sorting $2^0, 2^1, \ldots, 2^L$, and then testing $r, 3^{-1}r, 3^{-2}r, \ldots$ until one is found which is a power

of 2. Note that this will succeed even if L is not the actual length of the original bit string but is an upper bound on that length.

Let $n = \min\{a, b\}$ and

$$Y = \begin{pmatrix} r & s \\ 0 & 1 \end{pmatrix}, \quad \text{and} \quad U = (AB)^n A^{a-n} B^{b-n} = \begin{pmatrix} r & u \\ 0 & 1 \end{pmatrix}.$$

The goal is to replace several of the leading AB factors of U with BA, to transform it into Y. The basis for doing so is the observation that for all $W = \begin{pmatrix} w_{11} & w_{12} \\ 0 & 1 \end{pmatrix}$ we have

$$(AB)^j (BA) W = (AB)^j (AB) W + \begin{pmatrix} 0 & 6^j \\ 0 & 1 \end{pmatrix}.$$

Setting $t = s - u \pmod{p}$, suppose that $\mathbf{x} \in \{0, 1\}^n$ such that

$$\sum_{j=0}^{n-1} x_j 6^j \equiv t \pmod{p}. \tag{2.1}$$

Given such an $\mathbf{x}$, it follows that

$$\left(\prod_{j=0}^{n-1} (AB)^{1-x_j} (BA)^{x_j} \right) A^{a-n} B^{b-n} = U + \begin{pmatrix} 0 & t \\ 0 & 1 \end{pmatrix} = Y.$$

The *Subset Sum Problem* is the following: given a finite set $A = \{a_1, a_2, \ldots, a_n\}$ of integers and a target t, find a subset of A whose elements sum to t, if such a subset exists. It is well-known that the *density*, $n / \log_2(\max a_i)$, of an instance of this problem is an important parameter affecting the efficiency of various algorithms for solving it.

Lyubashevsky [2] considered a variant of this problem coined the Random Modular Subset Sum Problem (RMSSP): given a modulus M, a target t, and a set $A = \{a_1, a_2, \ldots, a_n\}$ of integers generated uniformly at random in $[0, M)$, find a subset of A whose elements sum to t modulo M. He defined the *density* of an instance of this problem to be $n / \log_2 M$; a *high-density* instance of this problem is one for which $n / \log_2 M > 1$.

The problem (2.1) we now need to solve is quite similar to the RMSSP, except that the set A is not generated at random; in this case, it is the set of residues of $6^0, 6^1, \ldots, 6^{n-1}$ modulo p. It is, however, a reasonable approximation to assume that it is uniformly random.

3 Solving the RMSSP

Note first the problem (2.1) we are faced with, considered as a RMSSP, is a very special instance, since the subset consists of small, consecutive powers of six. Nevertheless, we were unable to take advantage of this to find a more efficient solution.

Also note that if n is small compared to $\log_2 p$, there need not exist a solution; in fact, we have no proof that a solution exists even when $n > \log_2 p$, though when n is reasonably larger than $\log_2 p$ it certainly seems to be the case; and with values of n used in our experiments,

we never encountered any instances in which a solution was not found. In the sequel, we will assume that n is sufficiently large so that a solution does indeed exist. This is then a so-called *high-density* instance of the NMSSP, having $n > \log_2 p$. In fact, we will see later that we require n to be approximately larger than $2^{\sqrt{2\log_2 p}}$, so perhaps *very high-density* would be an appropriate descriptor.

There are algorithms in the literature for solving this problem; in particular, [2] proposes a method with sufficient asymptotic runtime, but a close inspection of the algorithm reveals that the implied constants in the memory/runtime estimates are too large to be practical in the current case.

Our approach is quite straightforward, and greedy in some sense. Loosely, the idea is to construct a set U which contains the canonical residues of $6^0, 6^1, \ldots, 6^{n-1}$ and $-t$. Subtract p from half of these at random, to obtain a set of integers in $(-p, p)$ for which about half are negative and half positive. Denote these integers by $a_0, a_1, \ldots, a_n$ with $a_n \equiv -t \pmod{p}$ and consider the matrix

$$\left(\begin{array}{c|c} & a_0 \\ & a_1 \\ I_{n+1} & \vdots \\ & a_n \end{array}\right).$$

At each step, we reduce the size of the largest (in absolute value) entry in the rightmost column (e.g., the ∞-norm), by finding another entry in that column with opposite sign such that

1. Adding the corresponding rows will not produce any non-$\{0, 1\}$ entries in the left block, and
2. the resulting value in the rightmost column has minimal possible absolute value, subject to the constraint above.

We repeat this process as many times as possible until it is no longer possible to shrink the ∞-norm of the vector on the right. At that point, we hope to find a row whose last two entries are 1 and 0 respectively. If that row is $\mathbf{x}$, it follows that

$$x_0 a_0 + \cdots + x_{n-1} a_{n-1} + 1(-t) \equiv 0 \pmod{p},$$

so that $t \equiv \sum_{j=0}^{n-1} x_j a_j$ as desired. Although we have no proof, the algorithm typically succeeds provided n is large enough. Below, we derive the estimate that n should be at least $2^{\sqrt{2\log_2 p}}$.

For a vector $\mathbf{x} = \langle x_0, x_1, \ldots x_n \rangle \in \mathbb{R}^{n+1}$, we denote the *support* of $\mathbf{x}$ by

$$\operatorname{supp}(\mathbf{x}) = \{j \in \{0, 1, \ldots, n\} \ : \ x_j \neq 0\}.$$

For an integer m and a positive integer p, we let $m \bmod p$ denote the nonnegative remainder of m divided by p. The algorithm we propose for solving the Modular Subset Sum Problem is as summarized below.

Algorithm 3.1
Input: *A list $a_0, a_1, \ldots, a_{n-1}$ of integers, a modulus p, and a target integer t.*
Output: *A vector $\mathbf{x} \in \{0,1\}^n$ such that $\sum x_j a_j \equiv t \pmod{p}$, or a failure message.*

1. *(Initialize) For $j = 0 \ldots n-1$, choose $\epsilon_j \in \{0,1\}$ at random and set $\widetilde{a_j} \leftarrow (a_j \mod p) - \epsilon_j p$, and set $\mathbf{u}_j \leftarrow \mathbf{e}_j$, the j-th standard basis vector in $\mathbb{R}^{n+1}$. Set $\widetilde{a_n} \leftarrow -t \bmod p$ and $\mathbf{u}_n \leftarrow \mathbf{e}_n$.*
2. *Find $i \in \{0, 1, \ldots, n\}$ such that $|\widetilde{a_i}|$ is maximal. Among all $j \in \{0, 1, \ldots, n\}$ for which $\operatorname{supp}(\mathbf{u}_j) \cap \operatorname{supp}(\mathbf{u}_i) = \emptyset$, find one with $|\widetilde{a_j} + \widetilde{a_i}|$ minimal. If no such j exists, goto Step 4.*
3. *Set $\mathbf{u}_i \leftarrow \mathbf{u}_i + \mathbf{u}_j$, $\widetilde{a_i} \leftarrow \widetilde{a_i} + \widetilde{a_j}$. Then if $\widetilde{a_i} = 0$ and the last coordinate of $\mathbf{u}_i$ is 1, goto Step 4. Otherwise, goto Step 2.*
4. *Find $k \in \{0, 1, \ldots, n\}$ such that $\widetilde{a_k} = 0$ and the last coordinate of $\mathbf{u}_k$ is 1. If no such k exists, output a failure message and terminate.*
5. *Set $\mathbf{x}$ to be the first n coordinates of $\mathbf{u}_k$, output $\mathbf{x}$ and terminate.*

For a proof of correctness, suppose that the algorithm terminates with an output vector $\mathbf{x}$. Let $u_{k,\ell}$ denote the ℓ-th coordinate of $\mathbf{u}_k$. First observe that at Step 2 we will always have

$$\sum_{\ell=0}^{n-1} u_{k,\ell} a_\ell + u_{k,n}(-t) \equiv \widetilde{a_k} \pmod{p},$$

for each $0 \le k \le n$. Therefore, if the algorithm terminates in Step 4 with some value of k, we will have

$$\sum_{\ell=0}^{n-1} u_{k,\ell} a_\ell + -t \equiv 0 \pmod{p},$$

so that the output vector $\mathbf{x}$ is indeed a solution.

Remark 3.2 *In the case when the algorithm outputs a failure message, it is possible to restart the algorithm and obtain a different outcome, because of the random choices made during initialization. If n is too small, it's possible that no choices of the ϵ_j will lead to success. When n is sufficiently large, it seems to succeed with nearly all choices of ϵ_j; but there seem to be borderline cases in between, when $n \approx 2\sqrt{2 \log_2 p}$; in these cases, restarting the algorithm enough times seems to eventually yield a solution. Our heuristic analysis will assume that the a_i's are uniformly distributed modulo p. However, if they are too clustered, we can precondition, multiplying all of the a_i's and t by a randomly chosen $r \in \mathbb{Z}_p^*$ prior to Step 1. Any resulting vector $\mathbf{x}$ will then still be a solution to the original system since it satisfies $\sum x_i r a_i \equiv rt \pmod{p}$.*

Remark 3.3 *In Step 2, finding i and j is considerably faster if $\widetilde{a_0}, \ldots, \widetilde{a_n}$ are kept sorted. For that reason, we store them in an AVL tree with key values $(\widetilde{a_k}, k)$. The operation $\widetilde{a_i} \leftarrow \widetilde{a_i} + \widetilde{a_j}$ in Step 3 is then performed by deleting the $(\widetilde{a_i}, i)$ node, performing the addition, and then re-inserting the node. Searches, insertions, and deletions each take $\mathcal{O}(\log n)$ operations.*

Remark 3.4 *Explicit storage of the* $\mathbf{u}_k$*'s requires* n^2 *bits, or* $(n^2/8)$ *bytes. Our implementation avoids this by not storing the rows explicitly. Instead, we store the row operations that were performed, and from these recover the* $\mathbf{u}_k$*'s on an as-needed basis. For this, we maintain an array* `OpList[0],...,OpList[n]` *of linked lists. Each time a row operation* $\mathbf{u}_i \leftarrow \mathbf{u}_i + \mathbf{u}_j$ *performed in Step 3, we append the node* (j, t) *to the linked list* `OpList[i]`*, where* t *is the current size of the linked list* `OpList[j]`*. With these data, we can reconstruct a vector* $\mathbf{u}_k$ *with* $\mathcal{O}(n)$ *operations as a consequence of the fact that whenever the operation* $\mathbf{u}_i \leftarrow \mathbf{u}_i + \mathbf{u}_j$ *is performed, the supports of* $\mathbf{u}_i$ *and* $\mathbf{u}_j$ *are disjoint.*

3.1 Heuristic analysis

Firstly, we make no claim of rigor for the analysis in this subsection. In particular, we make several unjustified assumptions, and use several estimates without bounding the error; so it should be considered as a 'back-of-the-envelope' estimate. However, the experimental data provided in Section 4 seem to fit the predictions made here reasonably well.

The first unjustified assumption we will make is that each time Step 2 is executed, the set of integers $A = \{\widetilde{a_0}, \ldots, \widetilde{a_n}\} - \{M\}$ are uniformly distributed in $[-M, M]$, where $M = \max\{|\widetilde{a_0}|, \ldots, |\widetilde{a_n}|\}$. We also unjustifiedly assume that the probability that $\operatorname{supp}(\mathbf{u}_j) \cap \operatorname{supp}(\mathbf{u}_i) = \emptyset$ is at least $1/2$.

Suppose now that $|\widetilde{a_i}| = M$. Then we expect about $n/2$ elements of A to have the opposite sign of $\widetilde{a_i}$, and we expect the largest absolute value of those to be about $(n/2)M/(n/2+1)$. So in Step 2 we expect a value of j to be found with

$$|\widetilde{a_j} + \widetilde{a_i}| \approx M - \frac{(n/2)M}{n/2+1} \approx \frac{2M}{n}.$$

In that case, we would expect that after the next n iterations of Step 2, we would have

$$\max\{|\widetilde{a_0}|, \ldots, |\widetilde{a_n}|\} \approx \frac{2M}{n}.$$

Initially, we have $M \approx p$. If every n iterations of Step 2 reduces $\max\{|\widetilde{a_0}|, \ldots, |\widetilde{a_n}|\}$ by a factor of $2/n$, then we would need to do bn iterations total to reduce this maximum to 1, where $b \approx \log_2 p / \log_2(n/2)$.

On the other hand, the algorithm will not be able to proceed once the probability that $\operatorname{supp}(\mathbf{u}_j) \cap \operatorname{supp}(\mathbf{u}_i) = \emptyset$ becomes too small; so long as this probability is at least $1/2$, the algorithm should be able to continue. If the average Hamming weight of $\mathbf{u}_0, \ldots, \mathbf{u}_n$ is about $w \in \mathbb{N}$, then the probability that two such randomly chosen vectors have disjoint support is about

$$\begin{aligned}
\frac{\binom{n-w}{w}}{\binom{n}{w}} &= \frac{(n-w)!(n-w)!}{n!(n-2w)!} \\
&= \left(\frac{n-w}{n}\right)\left(\frac{n-w-1}{n-1}\right)\cdots\left(\frac{n-w-(w-1)}{n-(w-1)}\right) \\
&= \left(1-\frac{w}{n}\right)\left(1-\frac{w}{n-1}\right)\cdots\left(1-\frac{w}{n-(w-1)}\right).
\end{aligned}$$

For large n and $w \leq \sqrt{n}$, this is approximately $(1 - w/n)^w$, which is about $e^{-w^2/n}$. If, for example, $w \leq (4/5)\sqrt{n}$ the probability that two such vectors are disjoint is at least $e^{-16/25} > 1/2$.

Finally, with each n iterations, we expect the average Hamming weight of the vectors $\mathbf{u}_0, \ldots, \mathbf{u}_n$ to roughly double. Initially this average weight is 1, so after bn iterations it will be about 2^b, and we need $2^b \leq \sqrt{n}$, so that $b \leq (1/2)\log_2 n$. Above we deduced that we need $b \approx \log_2 p / \log_2(n/2)$, and combining these results we find that we should have $n \geq 2^{\sqrt{2\log_2 p}}$. With $n \approx 2^{\sqrt{2\log_2 p}}$, the expected number of iterations is about $bn \approx (n/2)\log_2 n$.

3.2 Runtime

We will approximate the runtime of Algorithm 3.1 in terms of the number of $\mathbb{F}_p$ operations performed. Step 1 needs $\mathcal{O}(n)$ operations.

If the $\widetilde{a_k}$'s are stored in an AVL tree as suggested in Remark 3.3, we need $\mathcal{O}(\log n)$ operations to find i in Step 2. In searching for an appropriate value of j in Step 2, we find the largest (or smallest) value of $\widetilde{a}_k$ using the AVL tree and $\mathcal{O}(\log n)$ operations. Testing the disjoint support condition for a single value of j requires $\mathcal{O}(n)$ operations. As long as the average density of the vectors $\mathbf{u}_0, \ldots, \mathbf{u}_n$ is below $\sqrt{n}$, we expect Step 2 to check one or two values of j before finding an appropriate one. Therefore, we expect all but possibly the last execution of Step 2 to need $\mathcal{O}(n)$ operations, while the last one or two iterations will need $\mathcal{O}(n^2)$ operations. Step 3 needs only $\mathcal{O}(n)$ operations. Since the expected number of iterations is about $(n/2)\log_2 n$, it follows that all iterations of Steps 2 and 3 will need $\mathcal{O}(n^2 \log_2 n)$ operations total. Since this dominates the number of operations for Steps 4 and 5, we expect the total runtime to be $\mathcal{O}(n^2 \log_2 n)$, provided $n \approx 2^{\sqrt{2\log_2 p}}$.

4 Experimental results

The tables below summarize the results of experiments with the algorithm described in this paper. The algorithm was implemented in C, using the GMP library for arithmetic, and the experiments were carried out on a single core of an Intel i7-6700 processor at 3.4GHz, with 16GB of RAM. Each experiment consisted of the following:

1. Choose a random prime with the given number of bits,
2. Choose a random string consisting of N bytes,
3. Apply the hash function described in the introduction to obtain a hash value $(x, y) \in \mathbb{F}_p^2$,
4. Apply the algorithm described in this paper to find another string which hashes to (x, y).

In these tables, we also compare the number of iterations and time to obtain a solution with the heuristics obtained in the previous section. In doing so, n in that section corresponds

to the smaller of the number of zero bits and the number of one bits in the string; since the strings were generated uniformly at random, $n \approx N/4$ is a good approximation.

Let $\lambda(p) = 2^{-2+\sqrt{2\log_2 p}}$. This is the expected number of random bytes needed for our attack to work, according to the heuristics in the previous section. Table 1 compares, for randomly chosen primes p of various sizes and bit strings of different length $8N$, the average number of times Algorithm 3.1 had to be restarted before obtaining a solution. In all 1200 of these experiments a solution was obtained after restarting Algorithm 3.1 enough times.

$\log_2 p$	N	$N/\lambda(p)$	# experiments	avg. restarts
64	637	1.0	100	5.82
64	701	1.1	100	3.51
64	764	1.2	100	1.59
96	3708	1.0	100	2.62
96	4079	1.1	100	1.43
96	4450	1.2	100	1.07
128	16384	1.0	100	1.96
128	18023	1.1	100	1.17
128	19661	1.2	100	1.05
160	60664	1.0	100	1.62
160	66730	1.1	100	1.21
160	72797	1.2	100	1.05

Table 1: Experiments confirming the necessary length N in bytes for different primes p of different sizes. N is the number of bytes used in the experiments, while $\lambda(p) = 2^{-2+\sqrt{2\log_2 p}}$ is the predicted number of necessary bytes.

Table 2 compares the results of 490 experiments with the number of iterations predicted by our heuristic analysis, as well as the runtime bound predicted by that analysis.

5 Conclusion

We have demonstrated the the hash function proposed in [5] is not weak collision resistant, at least for inputs whose size is at least 1.9MB. Specifically, given such an input x we have been able to produce another input x' hashing to the same value, in about 13 hours on average. The attack presented here would be easily prevented in any number of non-algebraic ways; for example, by inserting an XOR operation once every 16 bits. However, this would defeat the main point of such a construction as it would seem to nullify the algebraic advantages. So it is an open question whether or not there is some 'nice' modification which would avoid this attack while still retaining the algebraic advantages.

$\log_2 p$	N	# experiments	$\overline{x}$	$\overline{x}/\rho(N)$	$\overline{t}$	$\overline{t}/\tau(N)$
64	764	100	18742	1.059	0.1	1×10^{-8}
96	4450	100	136981	1.090	0.7	2×10^{-9}
128	19661	100	694836	1.087	17.8	2.83×10^{-9}
160	72797	100	2870966	1.086	305.9	3.180×10^{-9}
192	237729	50	10221824	1.083	1770.7	1.578×10^{-9}
224	705871	20	32878864	1.087	16979.6	1.590×10^{-9}
256	1943805	20	96289524	1.082	47054.2	5.440×10^{-10}

Table 2: Experiments measuring the time of the attack for primes p of various sizes. N is the number of bytes used in the experiments. $\overline{x}$ is the average number of iterations per application of Algorithm 3.1 and $\rho(N) = 2N \log_2(4N)$ is the predicted number of iterations. The average time in seconds per experiment is $\overline{t}$ and $\tau(N) = N^2 \log_2 4N$ is a constant multiple of the asymptotic runtime bound heuristically derived in Section 3.2.

References

[1] Markus Grassl, Ivana Ilić, Spyros Magliveras, and Rainer Steinwandt. Cryptanalysis of the Tillich-Zémor hash function. *J. Cryptology*, 24(1):148–156, 2011.

[2] Vadim Lyubashevsky. On random high density subset sums. *Electronic Colloquium on Computational Complexity (ECCC)*, (007), 2005.

[3] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press Series on Discrete Mathematics and its Applications. CRC Press, Boca Raton, FL, 1997. With a foreword by Ronald L. Rivest.

[4] National Institute of Standards and Technology. FIPS PUB 180-4: Secure Hash Standard (SHS), August 2015. Available at `http://dx.doi.org/10.6028/NIST.FIPS.180-4`.

[5] Vladimir Shpilrain and Bianca Sosnovski. Compositions of linear functions and applications to hashing. *Groups Complex. Cryptol.*, 8(2):155–161, 2016.

[6] Jean-Pierre Tillich and Gilles Zémor. Hashing with SL_2. In *Advances in cryptology—CRYPTO '94 (Santa Barbara, CA, 1994)*, volume 839 of *Lecture Notes in Comput. Sci.*, pages 40–49. Springer, Berlin, 1994.