CHAMP: Cayley HAshing with Matrix Products

ALEXANDER DEMIN, ALEXEY OVCHINNIKOV, AND VLADIMIR SHPILRAIN

1. Introduction

Hash functions are easy-to-compute compression functions that take a variable-length input and convert it to a fixed-length output. Hash functions are used as compact representations, or digital fingerprints, of data and to provide message integrity. *Cryptographic hash functions* have many information-security applications, notably in digital signatures, message authentication codes, and other forms of authentication. Cryptographic hash functions should satisfy the following basic security requirements:

(1) *Collision resistance*: it should be computationally infeasible to find two different inputs that hash to the same output.

(2) *Preimage resistance* (sometimes called *non-invertibility*): it should be computationally infeasible to find an input which hashes to a specified output.

We note that collision resistance also implies *second preimage resistance*: it should be computationally infeasible to find a second input that hashes to the same output as a specified input.

A challenging problem is to determine mathematical properties of a hash function that would ensure (or at least, make it likely) that the requirements above are met.

A direction that has been gaining momentum lately is using a pair of elements, A and B, of a semigroup S, to hash the "0" and the "1" bit, respectively. Then a bit string is hashed to a product of elements in the natural way. For example, the bit string 1001011 will be hashed to the element $BAABABB$.

Since hashing a bit string this way represents a walk on the Cayley graph of the subsemigroup of S generated by the elements A and B, hash functions of this kind are often called *Cayley hash functions*. These hash functions have some useful functionalities that other hash functions (including the SHA family) do not have, see Section 2.

One might assume, in the spirit of "there is no free lunch", that the price to pay for these functionalities should be weaker security. However, up to date all known attacks on Cayley hashing proposals have been *ad hoc*, i.e., have been using specific "symmetry" in proposed pairs (A, B) of semigroup elements.

In this paper, we propose a pair (A, B) of matrices over $\mathbb{F}_p$ (for a safe prime p) with new properties that we argue should make the corresponding Cayley hash function immune to all known attacks.

The arrangement of the paper is as follows. We start by describing additional functionalities of Cayley hash functions in Section 2. In Section 3, we describe some of the previous proposals for platform (semi)groups as well as our own new proposal. In Section 4, we give a formal description of our hashing protocol and suggested parameters, as well as rationale

Research of the second and third authors was partially supported by a CUNY Institutional Research Grant Award.

for parameter selection. In Section 5, we give a Julia code for our protocol. In Section 6, we discuss security and possible attacks on our hash function. In Section 7, we give performance data and compare performance of our hash function to that of SHA-3. In the concluding Section 8, we report results of testing our hash function for pseudorandomness using the NIST statistical test suite.

2. Additional functionalities of Cayley hash functions

Cayley hash functions have some very useful properties that standard hash functions, like SHA family, do not have. We explain these properties below. In what follows, XY denotes concatenation of bit strings X and Y, and HM denotes the function that takes an arbitrary bit string to the corresponding product of matrices A and B.

I. Homomorphic property $HM(XY) = HM(X) \cdot HM(Y)$ for any bit strings X, Y.

II. Associativity property $HM(XYZ) = HM(XY) \cdot HM(Z) = HM(X) \cdot HM(YZ)$ for any bit strings X, Y, Z.

Why are these properties useful? In contrast with hash functions in the SHA family, Cayley hash functions have the following useful functionalities:

1. To hash a bit string X, one does not have to read the whole X first or even know the length of X; you can just hash X "as you go", bit by bit. You can even hash a stream of bits without knowing when it stops.

2. When an already hashed document X is amended by Y, one does not have to hash the whole amended document XY all over again, but rather hash just the amended part Y and then multiply the result by the hash of the original document because $HM(XY) = HM(X)HM(Y)$. Also note that one does not even have to know X to compute $HM(XY)$. Just knowing $HM(X)$ is enough.

Moreover, one can replace any middle part of a hashed document as well. Suppose the original document is a concatenation of 3 bit strings: XYZ, and suppose one wants to replace Y with Y'. Assuming X, Y, and Z were already hashed before to $HM(X)$, $HM(Y)$, and $HM(Z)$, respectively, we compute $HM(Y')$, and then $HM(XY'Z) = HM(X)HM(Y')HM(Z)$.

3. The property $HM(XY) = HM(X) \cdot HM(Y)$ allows for parallel computations in a natural way.

It looks like all these features apply to the function HM, the "intermediate" function, and not to the actual hash function H, see Section 3.1. Still, it is easy to see that all the features are preserved if the procedure of converting $HM(X)$ to $H(X)$ is *deterministic, efficient, and efficiently invertible*. Here is how it works.

For example, if an already hashed document X is amended by Y, here is how we can compute $H(XY)$ from $H(X)$ and $H(Y)$, respectively. First we recover $HM(X)$ and $HM(Y)$ from $H(X)$ and $H(Y)$. This is possible since we assumed that converting $HM(X)$ to $H(X)$ is efficiently invertible. Then we compute $HM(X) \cdot HM(Y) = HM(XY)$ and convert it to $H(XY)$.

3. Specific platform (semi)group

While the high-level idea of Cayley hashing is definitely appealing, the choice of the platform semigroup S and two elements $A, B \in S$ is crucial for security and efficiency. There have been many proposals based on matrix semigroups in $GL_2(\mathbb{F})$ for various fields $\mathbb{F}$, in particular for $\mathbb{F} = \mathbb{F}_p$. This is because Cayley graphs of 2-generator semigroups in $GL_2(\mathbb{F}_p)$ often have a large girth as was shown by several authors, see e.g. [3], [4], [7], [8]. Cayley graphs of (semi)groups in $GL_n(\mathbb{F}_p)$ with $n > 2$ have been considered, too (see e. g. [9]).

The first proposal of a Cayley hash function was due to Zémor [22]. The matrices used, considered over $\mathbb{F}_p$, were $A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, $B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$.

This proposal was successfully attacked in [19] (finding collisions) and recently in [10] (finding preimages). A specific property exploited in the attack in [19] was that this particular pair of matrices generates the whole semigroup of matrices from $SL_2(\mathbb{Z})$ having nonnegative entries. In addition, [10] used the fact that this pair of matrices generates the whole *group* $SL_2(\mathbb{Z})$.

The most cited proposal is what has become known as the Tillich-Zémor hash function [20]. Their matrices were $A = \begin{pmatrix} \alpha & 1 \\ 1 & 0 \end{pmatrix}$, $B = \begin{pmatrix} \alpha & \alpha+1 \\ 1 & 1 \end{pmatrix}$. These matrices are considered over a field defined as $R = \mathbb{F}_2[x]/(p(x))$, where $\mathbb{F}_2[x]$ is the ring of polynomials over $\mathbb{F}_2$, $(p(x))$ is the ideal of $\mathbb{F}_2[x]$ generated by an irreducible polynomial $p(x)$ of degree n (typically, n is a prime, $127 \le n \le 170$), and α is a root of $p(x)$.

The reason for selecting such a "fancy" field probably was to specifically avoid the attack in [19]. However, there were different attacks targeted at collision search in the Tillich-Zémor hash function, see [5], [15]. There were also attacks targeted at preimage search, see [14].

More recent proposals that use 2×2 matrices over $\mathbb{F}_p$ include [4] and [17]. Matrices of larger size were used in [1] and [9].

We, too, use a pair of 2×2 matrices, A and B, over $\mathbb{Z}$ and then reduce the entries modulo a large prime p to get matrices over $\mathbb{F}_p$, see Section 4.1. Rationale for selecting that particular pair is given in Section 4.2.

If p is a 256-bit prime, then a 2×2 matrix over $\mathbb{F}_p$ yields a 1024-bit string, the output of our hash function. If p is a 128-bit prime, then we get a 512-bit string as the output.

3.1. Converting output matrix to a bit string.

Denote by $HM(X)$ the matrix over $\mathbb{F}_p$ obtained while hashing a bit string X. This is not a final output of hashing because the final output has to be a bit string of a fixed length. Denote the final output of hashing by $H(X)$.

The procedure for converting $HM(X)$ to $H(X)$ has to be *deterministic, efficient, and efficiently invertible*. This is needed to preserve several useful features of the function $HM(X)$, see Section 2.

3.2. Converting a 2×2 matrix over $\mathbb{F}_p$ to a bit string of length 1024.

If p is a 256-bit prime, then each entry of $HM(X)$ can be naturally converted to a bit string of length 256, and thus a 2×2 matrix over $\mathbb{F}_p$ can be naturally, by concatenating 4 bit strings of length 256, converted to a bit string of length 1024.

However, the way we suggest here is slightly different. We first invert each nonzero entry of $HM(X)$ modulo p, and only after that convert to bit strings and concatenate.

This procedure is efficient; the slowest part is inverting modulo p. It is also deterministic and efficiently invertible.

3.3. **Converting a 2×2 matrix over $\mathbb{F}_p$ to a bit string of length 512.** If we want the output of a hash to be a 512-bit string, then we use a 128-bit prime p.

To convert $HM(X)$ to a bit string of length 512, one does the same thing as in Section 3.2.

4. Algorithm description

In the following description, we assume that p is a 256-bit prime and the output of the hash function has 1024 bits. For a 512-bit output, we use a 128-bit prime, but the algorithm itself is the same.

Given a bit string X (the input):

1. Going left to right along the bit string X, compute the product of copies of 2×2 matrices A and B over $\mathbb{F}_p$, where the matrix A corresponds to the "0" bit and the matrix B corresponds to the "1" bit. For example, if $X = 10011$, compute the product $HM(X) = BAABB$.

2. $HM(X)$ is a 2×2 matrix whose entries are integers modulo p. Compute the inverse (in $\mathbb{F}_p$) of each nonzero entry of $HM(X)$ and convert to a bit string of length 256. If an entry is 0, convert it to a string of 256 zeros. Denote the entries of the obtained matrix by h_{ij}.

3. Concatenate the 4 bit strings h_{ij} of length 256 obtained at Step 2 to a bit string of length 1024 in the following order: $h_{11}, h_{21}, h_{12}, h_{22}$. The obtained bit string is $H(X)$, the output of our hash function.

4.1. **Parameters.** Recall that for a 1024-bit output of our hash function we use a 256-bit prime p, whereas for a 512-bit output we use a 128-bit prime.

The 256-bit prime number p that we suggest is $p = 2^{256} - 36113$. This is a *safe prime*, i.e., $p - 1 = 2q$ for another prime q. This is actually the largest 256-bit safe prime.

The 128-bit prime number p that we suggest is $p = 2^{128} - 15449$. This is the largest 128-bit safe prime.

A particular pair of matrices (A, B) that we suggest is $A = \begin{pmatrix} -2 & 1 \\ 4 & -3 \end{pmatrix}$, $B = \begin{pmatrix} -5 & 2 \\ -6 & 2 \end{pmatrix}$.

An alternative pair is $A = \begin{pmatrix} -2 & 1 \\ 4 & -3 \end{pmatrix}$, $B = \begin{pmatrix} 6 & 4 \\ -2 & -1 \end{pmatrix}$.

For the rationale behind selecting these parameters, see Section 6 where security is discussed.

4.2. **Rationale for parameter selection.**

1. The reason for selecting p to be a safe prime is as follows. The determinant of both suggested matrices A and B is 2. Therefore, if there is a collision $u(A, B) = v(A, B)$, then $2^d = \pm 1$ in the multiplicative group $\mathbb{F}_p^*$ of $\mathbb{F}_p$. Here d is the difference in the lengths of u and v. Recall that the order of $\mathbb{F}_p^*$ is $(p - 1)$.

If $p = 2q+1$ is a safe prime, then $2^d = \pm 1$ in $\mathbb{F}_p$ implies that d is either 0 or at least q, i.e., very large. Thus, at least we have a guarantee that there are no collisions $u(A, B) = v(A, B)$ if $u(A, B)$ and $v(A, B)$ are not of the same length and are of length less than $\frac{p-1}{2}$.

2. One of the reasons for selecting matrices A and B as in Section 4.1 is for them to have determinant 2, thereby enforcing collision resistance, see above.

Another reason is that the suggested pairs of matrices should make the corresponding Cayley hash functions resistant to *length attack*, see the last two paragraphs of Section 6.2.

5. Implementations

We have implemented both CHAMP-1024 and CHAMP-512 in C code. In all implementations, we used the pair of matrices $A = \begin{pmatrix} -2 & 1 \\ 4 & -3 \end{pmatrix}$, $B = \begin{pmatrix} -5 & 2 \\ -6 & 2 \end{pmatrix}$.

Below is a basic code structure of CHAMP-1024 in Julia code that we provide for reference. Our fully functional optimized implementation of CHAMP-1024 (resp., CHAMP-512) has approximately 800 (resp., 600) lines of code.

Listing 1. A minimal reference implementation of CHAMP-1024 in Julia.

```
const p = big(2)^256 - 36113    # Using "big" multi-precision integers
const A = big.([-2  1;  4 -3;])
const B = big.([-5  2; -6  2;])

function champ_1024(
        msg::Vector{UInt8},     # Input:  array of bytes, message
        nbits::Int,             # Input:  integer, message bit length
        digest::Vector{UInt8}   # Output: 128-byte buffer
    )

    S = big.([1  0; 0  1])                          # Identity matrix
    for i in 0:nbits-1
        bit = (msg[div(i,8)+1] >> (7 - i%8)) & 1    # The i-th bit of message
        M   = iszero(bit) ? A : B
        S   = mod.(S * M, p)                        # Mat.-mul. modulo a prime
    end

    inv0(x) = iszero(x) ? big(0) : invmod(x, p)     # Invert nonzero entries
    S = inv0.(S)

    vals = [S[1,1], S[2,1], S[1,2], S[2,2]]
    for k in 1:4                                    # Write 128 output bytes
        for j in 0:31
            digest[32*(k-1)+j+1] = UInt8((vals[k] >> (8*(31-j))) & 0xff)
        end
    end
end
```

6. Security

Homomorphic properties of Cayley hash functions provide very useful additional functionalities (see Section 2), but intuitively, there should be a trade-off between additional functionalities and security (i.e., "there is no free lunch").

For example, if for an input Y, the matrix $HM(Y)$ is the identity matrix, then for any input X, one has $HM(XY) = HM(X)HM(Y) = HM(X)$, i.e., there is a collision. The

only way to avoid this problem is to only have very long Y such that $HM(Y) = I$, the identity matrix. In particular, if $A^k = I$ (or $B^k = I$), then k has to be very large. This is the case with our proposed parameters. Indeed, the determinant of both A and B is 2, so if $A^k = I$, then $2^k = 1$ in $\mathbb{F}_p$. Since p is a safe prime, i.e., $p = 2q + 1$ for another prime q, $2^k = 1$ implies here $k = q$, a 255-bit number. In other words, if $HM(Y) = I$, then Y is of length 2^{255}.

Still, the homomorphic property implies that if one collision was found, this would yield multiple collisions. Indeed, if $HM(Y) = HM(Z)$, then $HM(XY) = HM(XZ)$ for any bit string X. On the other hand, over the 35-year history of Cayley hash functions, collisions have been found only in some "highly symmetric" cases by using ad hoc attacks, see [19], [5], [15]. In the absence of general methods for finding collisions for Cayley hash functions, the focus is shifting to security properties relevant to preimage resistance, see Section 6.2 below.

6.1. **Security statements.** The following statements apply to classical as well as quantum computing settings.

1. There are provably no collisions $H(X) = H(Y)$ if the bit strings X and Y have different lengths and the difference between there lengths is less than 2^{255}.

If X and Y are of the same length, we do not have a theoretical proof of the infeasibility of finding collisions $H(X) = H(Y)$, but experimental evidence suggests that lower bounds on the girth of the Cayley graphs of matrix (semi)groups found in the literature ([3, 4, 7, 8]) are "too loose", i.e., the actual girth is actually huge in "most" cases.

2. Our hash function has passed all NIST pseudorandomness tests. This is a good (albeit indirect) evidence of preimage resistance, but a stronger security property is infeasibility of recovering the first (or the last) bit of a bit string X from the hash $H(X)$ with probability significantly larger than 0.5.

We claim that with our choice of parameters, it is infeasible to recover the first (or the last) bit of X from $H(X)$ with probability significantly larger than 0.5 if X has at least 1000 bits.

6.2. **Possible attacks.** For a while, the following property of semigroups of matrices over $\mathbb{F}_p$ has been considered an evidence of collision resistance. If two matrices A and B, when considered over $\mathbb{Z}$, generate a *free semigroup*, then there cannot be any collisions of the form $u(A, B) = v(A, B)$ modulo p unless at least some of the entries in $u(A, B)$ or $v(A, B)$ are greater than p.

Thus, several authors were looking for pairs of matrices A and B over $\mathbb{Z}$ whose products $u(A, B)$ would exhibit the slowest growth of maximal entries. Equivalently, the Cayley graph of the semigroup generated by A and B over $\mathbb{F}_p$ should have a large *girth*, see [3], [8].

The pair $A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, $B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ was considered in the seminal paper [22] where Cayley hash functions were introduced. The problem with this pair, however, is that these two matrices generate the whole semigroup of invertible 2×2 matrices over $\mathbb{Z}$ with non-negative entries. This often makes it feasible to find a preimage of a given matrix product by using a procedure similar to the Gauss elimination, only without divisions (see [19] for details).

In [4], the pair $A = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$, $B = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}$ was suggested. It was shown that, if p is a 256-bit prime, there cannot be any collisions of the form $u(A, B) = v(A, B)$ modulo p if the length of both $u(A, B)$ and $v(A, B)$ is less than 203.

However, this may not seem to be such a big selling point of the corresponding hash functions. First of all, it is understood that, in general (i.e., for any hash function), hashing very short bit strings is not preimage resistant since a preimage in that case can be found just by brute force, i.e. by hashing all possible bit strings up to a given length and comparing the results. In [11], the authors offered an efficient algorithm for finding collisions of length $O(\log p)$ for some Cayley hashing proposals based on matrices from $SL_2(\mathbb{F}_p)$. This is still rather short given that p is typically on the order of 2^{128} for 2^{256}. However, because of the homomorphic property $HM(XY) = HM(X)HM(Y)$, finding a short collision entails finding long collisions as well, which is why provably avoiding short collisions (as in [4]) is still important for Cayley hash functions. More about this in Section **??**.

Second and more interesting, having a pair (A, B) "too symmetric", as in the proposals of [22] and [4], might make the corresponding hash function vulnerable to a *length attack*, see [2]. The high-level idea is that multiplying (on the right) by the matrix A increases elements in the second column, whereas multiplying (on the right) by the matrix B increases elements in the first column. Thus, by looking at the product matrix, one can "locally" guess which matrix was the rightmost factor and therefore which bit was the rightmost in the input bit string. This attack was shown to have a success rate of over 50% for input bit strings of length up to a couple of thousand bits. Of course, this is nowhere near recovering a whole preimage, but since this is essentially the only known "generic" attack against Cayley hash functions based on matrices, it has to be addressed.

Informally, the reason why this attack may work is that the magnitude of entries in a matrix product increases monotonically. Then, given a matrix product, one can multiply it by A^{-1} or B^{-1} to see which one would decrease the magnitude of entries. Of course, there is no "magnitude" in $\mathbb{F}_p$, but this might work "locally" if p is large and the input bit string is relatively short.

There are two ways to foil such an attack. The first way is rather obvious: if both matrices A and B have determinant 2, then A^{-1} and B^{-1} have determinant 2^{-1}, which is a very large number in $\mathbb{F}_p$, so multiplying a matrix by either A^{-1} or B^{-1} will most likely increase the magnitude, not decrease.

Another way to foil the length attack is more sophisticated; it is based on an observation from [12]. The observation is: "on average" (in a precise sense), entries in a random product of matrices A and B (over $\mathbb{Z}$) grow as a linear combination of λ_1^n and λ_2^n, where λ_1 and λ_2 are eigenvalues of the "average matrix" $M = \frac{1}{2}(A + B)$. If at least one of λ_i is negative, then the growth (as a function of n) of their linear combination may not be monotone, and indeed it is not, as computer experiments show.

Now let us look at one of the pairs of matrices that we suggest: $A = \begin{pmatrix} -2 & 1 \\ 4 & -3 \end{pmatrix}$, $B = \begin{pmatrix} -5 & 2 \\ -6 & 2 \end{pmatrix}$. For this pair, $M = \frac{1}{2}(A + B) = \begin{pmatrix} -3.5 & 1.5 \\ -1 & -0.5 \end{pmatrix}$, and the eigenvalues of this matrix are $-2 \pm \frac{\sqrt{3}}{2}$, so they are both negative.

The alternative pair of matrices that we suggest: $A = \begin{pmatrix} -2 & 1 \\ 4 & -3 \end{pmatrix}$, $B = \begin{pmatrix} 6 & 4 \\ -2 & -1 \end{pmatrix}$. Here $M = \frac{1}{2}(A+B) = \begin{pmatrix} 2 & 2.5 \\ 1 & -2 \end{pmatrix}$, and the eigenvalues of this matrix are $\pm\sqrt{\frac{13}{2}}$, so one of them is positive and the other one is negative.

Figures 1 and 2 show how the "length" of a matrix product changes if one uses the matrices $A = \begin{pmatrix} -2 & 1 \\ 4 & -3 \end{pmatrix}$, $B = \begin{pmatrix} -5 & 2 \\ -6 & 2 \end{pmatrix}$. By the "length" here we mean the sum of the absolute values of the matrix entries. We see that the change is clearly not monotonic in this case.

7. Performance

We implemented our hash function in the 512-bit and 1024-bit output versions, each in two variants: reference and optimized implementations.

The optimized implementation incorporates two main performance boosters. First, since the entries of A and B are small integers, we accumulate the entries of several consecutive matrix products using native machine integer types before reducing modulo p, thus using fewer multiple-precision arithmetic operations. Second, we use a look-up table that stores the hash value corresponding to each possible byte (256 entries in total); equivalently, the table contains all possible products of eight generators (each being either A or B). This allows the implementation to process the input byte-by-byte rather than bit-by-bit.

In this section, we report experimental performance of the optimized implementation.

Setup	CHAMP-512	CHAMP-1024	SHA-512
(1) i5-8250U Windows	28.12 (MB/s)	17.77 (MB/s)	282.43 (MB/s)
(2) i9-13900 Linux	49.78 (MB/s)	44.03 (MB/s)	919.05 (MB/s)

Table 1. Throughput comparison across machine setups. Entries report the median MB/s for 100 MB input chunks.

Our benchmarking was performed in two different setups representing popular platforms:

(1) An x86-64 laptop running Windows 10 on an Intel i5-8250U CPU, with 1.80 GHz processor frequency reported by the system for the benchmark session. The CHAMP-512 and CHAMP-1024 were compiled from source using GCC 11.1.0 with the flags `-O3 -DNDEBUG`. The SHA-512 baseline uses the compiled `bcrypt` library accessible via the Win32 CNG interface. The CHAMP-512 and CHAMP-1024 implementations run in single-threaded mode.

(2) An x86-64 workstation running Linux 6.1.0-1029-oem on an Intel i9-13900 CPU. No fixed processor frequency was recorded for the benchmark session; `/proc/cpuinfo` snapshots taken during the benchmark showed per-core instantaneous readings up to about 4.60 GHz on active cores. The implementations were compiled with GCC 11.4.0 using the same release flags `-O3 -DNDEBUG`. The CHAMP benchmarks again use our optimized single-threaded C implementations. The SHA-512 baseline uses the OpenSSL software interface through `libcrypto`.

Performance was measured by repeatedly hashing a 100 MB chunk of data and collecting the running times. Table 1 reports the resulting throughput in MB/s.

We observe that CHAMP-512 is faster than CHAMP-1024, in part because it uses 128-bit integers instead of the 256-bit ones for implementing arithmetic operations in $\mathbb{F}_p$.

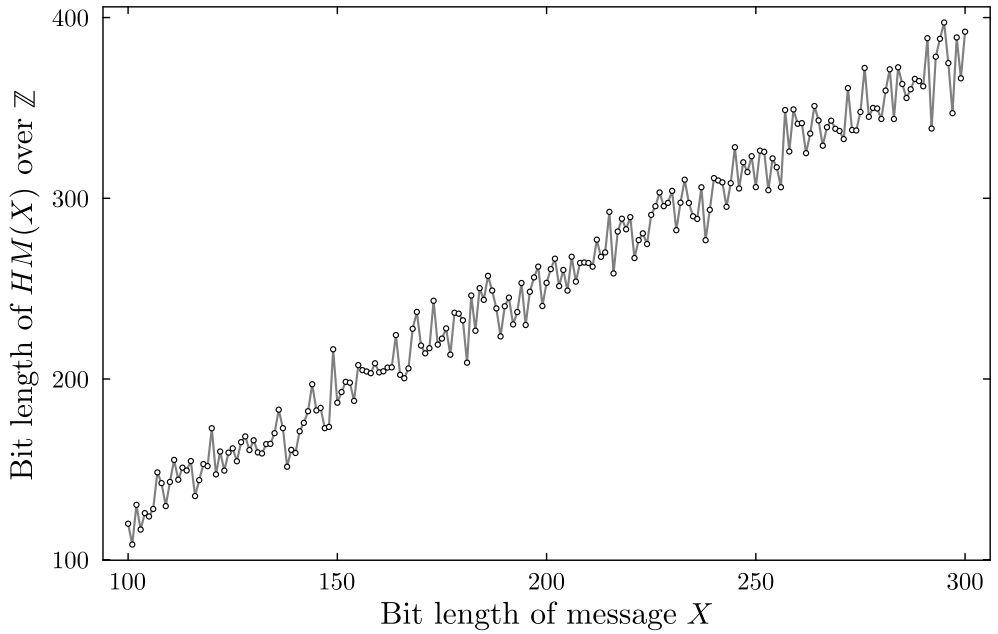

FIGURE 6.1. Bit length of the sum of absolute values of the entries of $HM(X)$, the latter computed over $\mathbb{Z}$ without modular reduction, as a function of the bit length of the input X. The inputs X are chosen uniformly at random.

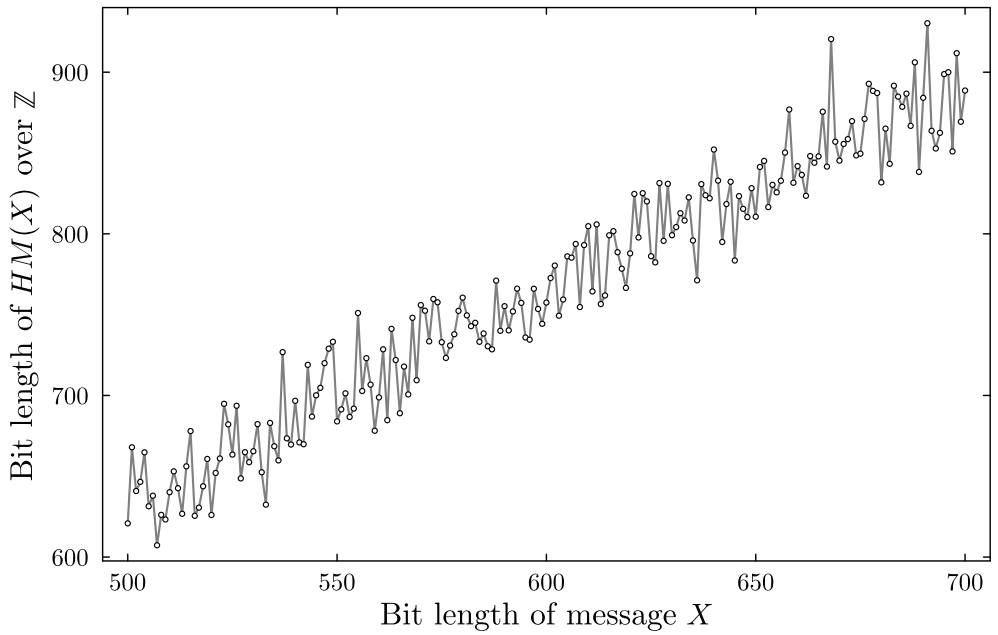

FIGURE 6.2. Bit length of the sum of absolute values of the entries of $HM(X)$.

As for comparing CHAMP-512 to SHA-512, Table 1 shows that CHAMP-512 appears to be 10 to 20 times slower. However, due to the homomorphic property (see Section 2), the input can be split in, say, 8 parts to be hashed simultaneously, and then the resulting 8 matrices would be multiplied together before converting to a 512-bit string. This would speed up computation almost by the factor of 8.

8. NIST pseudorandomness tests

We have tested the outputs of our hash function for pseudorandomness using the NIST statistical test suit [13] for both 512-bit and 1024-bit output cases.

We describe the experimental setup for the case of 1024-bit output; the 512-bit one is similar. For a positive number X smaller than 2^{1024}, we denote by $H(X)$ the result of applying our hash function to the 1024-bit little-endian binary representation of X. We compute $H(1), H(2), \ldots, H(N)$ and concatenate the results in a single bit string of length $1024N$ bits. We pick N sufficiently large so that the length of the resulting bit string is at least 100 MB.

We run the tests with the default values of parameters, using 100 bit streams each 10^6 bits in length. The default values of parameters were:

```
[1] Block Frequency Test - block length(M):          128
[2] NonOverlapping Template Test - block length(m): 9
[3] Overlapping Template Test - block length(m):     9
[4] Approximate Entropy Test - block length(m):      10
[5] Serial Test - block length(m):                   16
[6] Linear Complexity Test - block length(M):        500
```

Testing results for both pairs of matrices suggested in Section 4.1 are collected in four text files. For each pair of matrices, tests were run for both 512-bit and 1024-bit output, which is why we have four files with test results altogether.

According to these results, our hash function $H(X)$ has successfully passed all NIST pseudorandomness tests, for both pairs of matrices suggested in Section 4.1.

9. Open problems

As we mentioned in Section 6.2, it was shown in [4] that the girth of the Cayley graph of the semigroup generated by $A = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$ and $B = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}$ (considered over $\mathbb{F}_p$) is at least 203 if p is a 256-bit prime.

The fact that the corresponding Cayley hash function has not been successfully attacked up to date probably means that the actual girth of the Cayley graph here is a lot larger than 203. Therefore we ask:

Problem 9.1. *What is the actual girth of the Cayley graph of the semigroup generated by* $A = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$ *and* $B = \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix}$ *(considered over* $\mathbb{F}_p$*) if* $p = 2^{256} - 36113$*?*

The prime $p = 2^{256} - 36113$ is a *safe prime*, i.e., $p - 1 = 2q$ for another prime q. This is actually the largest 256-bit safe prime.

We note that if negative entries are allowed, the situation becomes a lot more mysterious. We ask:

Problem 9.2. *What is the actual girth of the Cayley graph of the semigroup generated by* $A = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$ *and* $B = \begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix}$ *(considered over* $\mathbb{F}_p$*) if* $p = 2^{256} - 36113$*?*

With the help from Nicola Guglielmi [6] we were able to get a lower bound on the girth of the Cayley graph in this case, following the method used in [4]. The lower bound is 270 and again, we believe that the actual girth is a lot larger.

We note that since p in these problems is a 256-bit number, a 2×2 matrix over $\mathbb{F}_p$ is representable by a bit string of 1024 bits. Therefore, by the pigeonhole principle, an upper bound on the girth of the Cayley graph in this case is 1025+1025=2050.

As for pairs of matrices we suggest in Section 4.1, we could, of course, ask the same question, but there are currently no methods to even try to attack such a problem for those matrices.

References

[1] Y. Aikawa, H. Jo, S. Satake, *Left-right Cayley hashing: a new framework for provably secure hash functions*, Math. Cryptology **3** (2023), 53—65.

[2] Y. Antolin, M. I. Gonzalez Vasco, I. M. Urrutia, *Length attacks on matrix group hashing*, preprint.

[3] J. Bourgain, A. Gamburd, *Uniform expansion bounds for Cayley graphs of* $SL_2(\mathbb{F}_p)$. Ann. of Math. (2) **167** (2008), 625–642.

[4] L. Bromberg, V. Shpilrain, A. Vdovina, *Navigating in the Cayley graph of* $SL_2(\mathbb{F}_p)$ *and applications to hashing*, Semigroup Forum **94** (2017), 314–324.

[5] M. Grassl, I. Ilić, S. Magliveras, R. Steinwandt, *Cryptanalysis of the Tillich-Zémor hash function,* J. Cryptolgy **24** (2011), 148—156.

[6] N. Guglielmi, informal communication.

[7] S. Han, A. M. Masuda, S. Singh, J. Thiel, *Maximal entries of elements in certain matrix monoids*, Integers **20** (2020), paper No. A31.

[8] H. A. Helfgott,*Growth and generation in* $SL_2(\mathbb{Z}/p\mathbb{Z})$ Ann. of Math. (2) **167** (2008), 601–623.

[9] C. Le Coz, C. Battarbee, R. Flores, T. Koberda, D. Kahrobaei, *Post-quantum hash functions using* $SL_n(\mathbb{F}_p)$, Adv. Math. Commun. **19** (2025), 996–1009.

[10] E. McKemmie, A. Srivastava, *Preimages for Zémor's Cayley hash function*, preprint. https://arxiv.org/abs/2511.15842

[11] C. Mullan and B. Tsaban, SL_2 *homomorphic hash functions: Worst case to average case reduction and short collision search*, Des. Codes Cryptogr. 81 (2016), 83–107.

[12] N. Nabahi, V. Shpilrain, *Easy estimates of Lyapunov exponents for random products of matrices*, preprint. `https://arxiv.org/abs/2509.18944`

[13] National Institute of Standards and Technology - NIST , *NIST Statistical Test Suite*, 2010. `http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html`

[14] C. Petit, J.-J. Quisquater, *Preimages for the Tillich-Zémor hash function*, in: SAC 2010. LNCS, **6544** (2011), 282–301.

[15] C. Petit, J.-J. Quisquater, J.-P. Tillich, G. Zémor, *Hard and easy components of collision search in the Zémor-Tillich hash function: new attacks and reduced variants with equivalent security*, Topics in cryptology-CT-RSA 2009, Lecture Notes in Comput. Sci. **5473** (2009), 182–194.

[16] V. Shpilrain, *Girth of the Cayley graph and Cayley hash functions*, London Math. Soc. Newsletter **514** (2025), 27–30.

[17] V. Shpilrain, B. Sosnovski, *Compositions of linear functions and applications to hashing*, Groups, Complexity, Cryptology **8** (2016), 155–161.

[18] V. Shpilrain, B. Sosnovski, *Cayley hashing with cookies*, in: Future of Information and Communication Conference (FICC 2025), Springer Lecture Notes in Networks and Systems **1284** (2025), 706–718.

[19] J.-P. Tillich and G. Zémor, *Group-theoretic hash functions*, in Proceedings of the First French-Israeli Workshop on Algebraic Coding, Lecture notes Comp. Sci. **781** (1994), 90–110.

[20] J.-P. Tillich and G. Zémor, *Hashing with SL_2*, in CRYPTO 1994, Lecture Notes Comp. Sci. **839** (1994), 40–49.
[21] J. von zur Gathen, I. E. Shparlinski, *Generating safe primes*, J. Math. Cryptol. **7** (2013), 333–365.
[22] G. Zémor, *Hash functions and graphs with large girths*, in: Eurocrypt'91, Lecture Notes in Comput. Sci. **547** (1991), 508–511.

Laboratoire d'informatique de l'École polytechnique, LIX, UMR 7161, CNRS, 1 rue Honoré d'Estienne d'Orves, 91120 Palaiseau, France
Email address: alexander.demin@lix.polytechnique.fr

Department of Mathematics, Queens College, City University of New York, Queens, NY 11367
Email address: alexey.ovchinnikov@qc.cuny.edu

Department of Mathematics, The City College of New York, New York, NY 10031
Email address: shpilrain@yahoo.com