COMPOSITIONS OF LINEAR FUNCTIONS AND APPLICATIONS TO HASHING

VLADIMIR SHPILRAIN AND BIANCA SOSNOVSKI

ABSTRACT. Cayley hash functions are based on a simple idea of using a pair of (semi)group elements, A and B, to hash the 0 and 1 bit, respectively, and then to hash an arbitrary bit string in the natural way, by using multiplication of elements in the (semi)group. In this paper, we focus on hashing with linear functions of one variable over $\mathbb{F}_p$. The corresponding hash functions are very efficient. In particular, we show that hashing a bit string of length n with our method requires, in general, at most $2n$ multiplications in $\mathbb{F}_p$, but with particular pairs of linear functions that we suggest, one does not need to perform any multiplications at all. We also give explicit lower bounds on the length of collisions for hash functions corresponding to these particular pairs of linear functions over $\mathbb{F}_p$.

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. Basic requirements are well known:

(1) *Preimage resistance* (sometimes called *non-invertibility*): it should be computationally infeasible to find an input which hashes to a specified output;
(2) *Second pre-image resistance*: it should be computationally infeasible to find a second input that hashes to the same output as a specified input;
(3) *Collision resistance*: it should be computationally infeasible to find two different inputs that hash to the same output.

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.

An interesting direction worth mentioning is constructing hash functions that are provably as secure as underlying assumptions, e.g. as discrete logarithm assumptions; see [4] and references therein. These hash functions however tend to be not very efficient. For a general survey on hash functions we refer to [7].

Another direction, relevant to the present paper, is using a pair of elements, A and B, of a semigroup S, such that the Cayley graph of the semigroup generated by A and B has a large girth and therefore there are no short relations in the semigroup, meaning there are no short collisions for the relevant hash function. Probably the

Research of the second author was partially supported by the NSF grant CNS-1117675 and by the ONR (Office of Naval Research) grant N000141210758.

most popular implementation of this idea so far is the Tillich-Zémor hash function [15]. We refer to [10] and [12] for a more detailed survey on Cayley hash functions.

The Tillich-Zémor hash function, unlike functions in the SHA family, is *not* a block hash function, i.e., each bit is hashed individually. More specifically, the "0" bit is hashed to a particular 2×2 matrix A, and the "1" bit is hashed to another 2×2 matrix B. Then a bit string is hashed simply to the product of matrices A and B corresponding to bits in this string. For example, the bit string 1000110 is hashed to the matrix BA^3B^2A.

Tillich and Zémor use matrices A, B from the group $SL_2(R)$, where R is a commutative ring (actually, a field) defined as $R = \mathbf{F}_2[x]/(p(x))$. Here $\mathbf{F}_2$ is the field with two elements, $\mathbf{F}_2[x]$ is the ring of polynomials over $\mathbf{F}_2$, and $(p(x))$ is the ideal of $\mathbf{F}_2[x]$ generated by an irreducible polynomial $p(x)$ of degree n (typically, n is a prime, $127 \le n \le 170$); for example, $p(x) = x^{131} + x^7 + x^6 + x^5 + x^4 + x + 1$. Thus, $R = \mathbf{F}_2[x]/(p(x))$ is isomorphic to $\mathbf{F}_{2^n}$, the field with 2^n elements.

Then, the matrices A and B are:

$$A = \begin{pmatrix} \alpha & 1 \\ 1 & 0 \end{pmatrix}, \qquad B = \begin{pmatrix} \alpha & \alpha+1 \\ 1 & 1 \end{pmatrix},$$

where α is a root of $p(x)$.

This particular hash function was successfully attacked in [6] and [11]; see also [8] for a more general attack approach. It should be pointed out though that the general attack in [8] works in super-polynomial time in the size of n; in particular, it becomes infeasible already for the value $n = 131$ suggested in [15]. The attack in [6] is more special, targeted specifically at the hash function in [15] (in particular, they use a known result about a worst-case behavior of the Euclidean algorithm in $\mathbf{F}_2[x]$), and it does actually find short collisions for various irreducible polynomials $p(x)$, including polynomials $p(x)$ of a rather high degree, like $n = 2039$. Again, this attack is specific to (some pairs of) matrices over a Galois field of characteristic 2. A more general attack in [8] works in super-polynomial time in the size of n, so it is infeasible for a generic irreducible polynomial $p(x)$ of degree $n > 100$, say. To be fair though, we should mention that with $p(x)$ of a high degree, efficiency of the Tillich-Zémor hash function will fall behind that of hash functions in the SHA family; the latter are the prevailing standard for hash functions, at least in the United States. We also note that the attack reported in [11] is rather different: it provides a technique for finding preimages, rather than collisions, for the Tillich-Zémor hash function.

In a recent paper [1], the authors suggested other pairs of matrices, of the form $\begin{pmatrix} 1 & k \\ 0 & 1 \end{pmatrix}$ and $\begin{pmatrix} 1 & 0 \\ k & 1 \end{pmatrix}$, $k \ge 2$, with the idea that since these matrices generate a free monoid over $\mathbb{Z}$, there cannot be any short relations between them over $\mathbb{F}_p$.

In this paper, we also offer hashing with a pair of 2×2 matrices, but these matrices generate a (semi)group isomorphic to the (semi)group (with respect to composition) of linear functions of one variable over $\mathbb{F}_p$. The corresponding hash functions are very efficient; the time complexity of hashing a bit string of length n with our method is determined by performing at most $2n$ multiplications and

about $2n$ additions in $\mathbb{F}_p$ (see our Section 4 for more details). In fact, if one uses linear functions with small coefficients at x, then multiplications can be avoided altogether. This is because, say, $2x = x+x$ and $3x = x+x+x$. Reductions modulo p after multiplication by 2 or 3 do not have to involve any multiplications (or inversions) either. Thus, hashing with such a pair of linear functions is about as efficient as it gets.

The input bit string for our hash function can have an arbitrary length, while the output (with our suggested parameters) is a 256-bit number (the constant term of the composite linear function). If we compare this to the Tillich-Zémor hash function and to the hash function in [1], we see that in these previous proposals, if one uses a ground field of size 2^{256}, then the size of a hash will be 1024 bits, versus 256 bits in our case, which gives our hash function another advantage as far as performance is concerned.

In Section 2, we give explicit lower bound on the length of collisions for the hash functions corresponding to some particular pairs of linear functions over $\mathbb{F}_p$. One particular pair is $f(x) = 2x+3$, $g(x) = 3x+1$, and we show in Section 2 that if two bit strings have the same hash, then the length of at least one of the bit strings is at least $\log_3 p$, which gives a lower bound of 162 if p is a 256-bit number.

In Section 3, we discuss security of our hash function, and in Section 4 we give some performance results and compare them to performance of SHA-256. Here we just mention that a serious advantage (in terms of performance) of Cayley hash functions, including our function, over hash functions in the SHA family is that computing any Cayley hash function H can be easily parallelized due to the homomorphic property $H(AB) = H(A)H(B)$ and the associativity property $H(ABC) = H(AB)H(C) = H(A)H(BC)$ for any bit strings A, B, C. (Here AB means a simple concatenation of A and B.) Thus, one can split a bit string in several pieces, compute the hash of each piece separately, and then multiply out the hashes.

2. Proposed hash function

Let $f(x) = ax+b$ and $g(x) = cx+d$ be two linear functions with integer coefficients. The semigroup (under composition) generated by these two functions is isomorphic to the semigroup of matrices generated by the following 2×2 matrices (assuming that the composition $f(x)g(x)$ is interpreted so that $g(x)$ is applied first):

$$A = \begin{pmatrix} a & b \\ 0 & 1 \end{pmatrix}, \qquad B = \begin{pmatrix} c & d \\ 0 & 1 \end{pmatrix}.$$

It follows from the results of [3] that the semigroup generated by $f(x)$ and $g(x)$ is free if $f(x)$ and $g(x)$ do not commute and $a, c \geq 2$. Thus, for example, $f(x) = 2x+3$ and $g(x) = 3x+1$ generate a free semigroup since these two functions do not commute: $f(x)g(x) = 6x+5$, while $g(x)f(x) = 6x+10$. Moreover, what is actually proved in [3] is that the *constant terms* of two different elements of the semigroup generated by $f(x)$ and $g(x)$ are different if $f(x)$ and $g(x)$ do not commute and $a, c \geq 2$.

If the coefficients of linear functions are now considered as elements of some $\mathbb{F}_p$ rather than $\mathbb{Z}$, then there cannot be an equality of two different semigroup words in $f(x)$ and $g(x)$ unless at least one of the coefficients in at least one of the two words is $\geq p$. Thus, we propose:

Proposed hash function. Let the '1' bit be hashed to $f(x) = 2x+3$ and the '0' bit be hashed to $g(x) = 3x+1$. To find the hash of a given bit string B, one computes, over $\mathbb{F}_p$, the composition $P(x)$ of copies of $f(x)$ and $g(x)$ corresponding to the bits in B, and then takes the constant term of $P(x)$. In particular, the hash of B is an element of $\mathbb{F}_p$.

Now we give a lower bound on the minimum length of bit strings where a collision may occur.

Proposition 1. *Let the '1' bit be hashed to $f(x) = 2x+3$ and the '0' bit be hashed to $g(x) = 3x+1$. If two bit strings U and V hash to the same number, then the length of either U or V is at least* $\log_3 p$.

Proof. Suppose the length of U is n, and the length of V is $\leq n$. If $L(x) = rx+s$ is the hash of U, then it is easy to see that both coefficients r and s are less than or equal to 3^n. Therefore, if $3^n < p$, the hashes of U and V cannot be equal because otherwise they would be equal also over $\mathbb{Z}$, which is impossible.

Thus, if the longer of the two bit strings has length $< \log_3 p$, their hashes cannot be equal over $\mathbb{F}_p$. □

For example, if p is on the order of 2^{256}, the above hash function cannot have collisions unless the length of at least one of the colliding bit strings is at least 162.

3. Security

Not many attacks on Cayley hash functions are known. Probably the most "generic" one is the "lifting attack" [14]. The idea is to find a preimage of a given hash in the ambient free (semi)group by "splitting out" one (semi)group generator at a time, so that the "size" of the result would decrease at every step. In our context, where the hash is the constant term of a linear function $P(B) = rx+s$ over $\mathbb{F}_p$, one would lift $P(B)$ to a linear function $Rx+S$ over $\mathbb{Z}$ (i.e., R arbitrary, but $S = s+kp$ for some $k \in \mathbb{Z}$) and try to multiply it by either $f^{-1}(x)$ or $g^{-1}(x)$ to decrease one or both coefficients (or, perhaps, to decrease their sum). However, there are two major obstacles that basically make this attack void. The main obstacle is that lifting itself in our situation is by no means unique, and there is no way to tell just by inspection which one is a "good" lifting. This is in sharp contrast with the situation considered in [14] where a "good" lifting can be detected by inspection. The only necessary condition for a lifting to be "good" in our situation is that the coefficient at x should be of the form $2^m 3^n$, but the only condition (visible by inspection) on the constant term S is that either $S-1$ is divisible by 3 or $S-3$ is divisible by 2, which leaves a lot of possibilities for lifting.

The other obstacle is that splitting out a generator in this case is not unique either, because at every step of the procedure, one would have the constant term C

such that both $C-1$ is divisible by 3 and $S-3$ is divisible by 2 with non-negligible probability, thus creating a tree of possible reduction sequences, where only one sequence is correct (assuming that the lifting was "good" to begin with, so that there actually *is* a correct sequence). Thus, even if the attacker was lucky to pick a "good" lifting, he will still have to search over exponentially many (in the length of an input bit string) possible reduction sequences. To be fair though, this problem is relatively insignificant compared to the problem of finding a "good" lifting for the constant term s.

Another known attack on a Cayley hash function is that in [6], but there a collision for the Tillich-Zémor hash function [15] was found using an algorithm specific to Galois fields of characteristic 2. A more general attack in [8] uses a "birthday paradox" kind of argument and involves a "brute force" search over all bit strings of a length depending on the size of the ground field. In our situation, since we have a solid lower bound of 162 for the minimum length of colliding bit strings (if p is on the order of 2^{256}), a similar approach would involve a "brute force" search over bit strings of length about 80, which is computationally infeasible, so the "birthday paradox" reduction is simply not enough to make the attack feasible with our suggested size of p.

We have also experimentally verified statistical properties of our hash function using NIST statistical test suite, see below.

3.1. **NIST statistical test suite results.** To evaluate how pseudorandom our hash functions is, we applied the NIST Statistical Test Suite [9] for randomness to a sequence of hash values in binary form.

The NIST Statistical Test Suite is a package that includes 15 types of tests, each with a suitable metric needed to investigate the degree of randomness for binary sequences produced by cryptographic random generators.

To meet the minimum recommendations [13] on the length of the bit strings tested, the prime $p = 139948 \cdot 151^{139948} + 1$ (see [2]) with bit length of 1,013,018 was used as parameter for the hash function to generate bit strings. We tested 100 of such bit strings. The level of significance applied used in the suite is $\alpha = 0.01$.

Table 1 below presents the statistical properties of the hash values as reported by the NIST test suite. According to NIST documentation, a pass rate of 96% is acceptable.

We note that the bit strings tested passed every test in the NIST Statistical Test Suit. They also passed each individual non-overlapping templates tests, random excursions and random excursions variant tests performed, although we only present here the averaged results for those tests.

4. PERFORMANCE

Recall that computing any Cayley hash function H can be easily parallelized due to the homomorphic property $H(AB) = H(A)H(B)$ and the associativity property $H(ABC) = H(AB)H(C) = H(A)H(BC)$ for any bit strings A, B, C. (Here AB means a simple concatenation of A and B.) Thus, one can split a bit string in several

pieces, compute the hash of each piece separately, and then multiply out the hashes (in our situation, multiplication is composition of (linear) functions).

A nice additional feature of our Cayley hash function, which is based on linear functions, is that parallelization in this case can be taken a little further. Since any linear function is determined by its values at two points, we can run evaluation of the composite linear function at two points in parallel, thus reducing the computation time by half to begin with.

To compare performance of our hash function (with $f(x) = 2x+3$ and $g(x) = 3x+1$) to that of other hash functions, we first recall that the input bit string in our situation can have an arbitrary length, while the output (with our suggested parameters) is a 256-bit number or, equivalently, a 256-bit string. That means, the output is of the same length as it is for SHA-256, so below we compare performance of our hash function to that of SHA-256, see [5].

According to [5], SHA-256 hashes approximately 111 MiB/second (MiB stands for mebibyte, and 1 MiB = 2^{20} bytes) and so this is roughly 10^8 bytes per second. Our proposed hash function, with $p = 2^{256} - 1053$, hashes 10^8 bytes in about 1.2 seconds without any optimization or parallelization. The tests were performed on an Intel Core i7 3.4GHz computer with 8 GB of RAM using Python 3.4. With a standard 4-core desktop computer, parallelizing computation (see the beginning of this section) using all 4 cores will therefore make hashing of long bit strings with our hash function (again, without any optimization) almost 4 times as fast as with SHA-256.

To conclude this section, we emphasize once again one of the reasons for efficiency of our hash function. Even though all computation is done in $\mathbb{F}_p$, in our situation we can altogether avoid multiplications during reductions modulo p. This is because coefficients in our linear functions $f(x) = 2x+1$ and $g(x) = 3x+1$ are quite small, so when we multiply an integer $x < p$ by 2 or 3 and it becomes greater than p, all we have to do to reduce it modulo p is to subtract p or $2p$. Therefore, with coefficients at x as small as 2 and 3, multiplications (and inversions) can be avoided altogether because multiplication by 2 is the same as addition, and multiplication by 3 amounts to two additions. Thus, with our suggested parameters, one needs to perform between $3n$ and $4n$ additions and no multiplications to hash a bit string of length n.

It may seem that one inversion is still needed to recover a linear function from its values at two points, but since the choice of the two points x_1, x_2 is up to us, we can choose them so that $x_2 - x_1 = 1$, so that inversion is actually not needed.

5. CONCLUSIONS

• We have described a very efficient Cayley hash function where "0" and "1" bits are hashed by linear functions over $\mathbb{F}_p$. With particular suggested pair of linear functions $f(x) = 2x+3, g(x) = 3x+1$, one needs to perform between $3n$ and $4n$ additions in $\mathbb{F}_p$ and no multiplications (or inversions) to hash a bit string of length n.

• Computation of any Cayley hash function can be easily parallelized. If p is on the order of 2^{256}, then the output of our hash function is of size 256 bits, as in SHA-256. At the same time, already on a 2-core processor our hash function outperforms SHA-256, even without any optimization.

• If p is on the order of 2^{256}, then with the suggested pair of linear functions, our hash function does not have collisions unless the length of at least one of the colliding bit strings is at least 162.

• Our hash function has successfully passed all the pseudorandomness tests in the NIST Statistical Test Suit.

REFERENCES

[1] L. Bromberg, V. Shpilrain, A. Vdovina, *Navigating in the Cayley graph of $SL_2(\mathbb{F}_p)$ and applications to hashing*, Semigroup Forum, to appear. http://arxiv.org/abs/1409.4478

[2] C. Caldwell, *The Primes Pages*. https://primes.utm.edu

[3] J. Cassaigne, T. Harju, J. Karhumki, *On the undecidability of freeness of matrix semigroups*, Internat. J. Algebra Comput. **9** (1999), 295–305.

[4] S. Contini, A. K. Lenstra and R. Steinfeld, *VSH, an Efficient and Provable Collision Resistant Hash Function*, in: Eurocrypt 2006, Lecture Notes Comp. Sci. **4004** (2006), 165–182.

[5] W. Dai, *Crypto++ 5.6.0 benchmarks*, http://www.cryptopp.com/benchmarks.html

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

[7] A. Menezes, P. van Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

[8] C. Mullan and B. Tsaban, *Short collision search in arbitrary SL_2 homomorphic hash functions,* preprint, http://arxiv.org/abs/1306.5646

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

[10] C. Petit, *On graph-based cryptographic hash functions*, PhD thesis, 2009.

[11] C. Petit, J. Quisquater, *Preimages for the Tillich-Zémor hash function,* in: SAC 10, Lecture Notes Comp. Sci. **6544** (2010), 282-301.

[12] C. Petit and J.-J. Quisquater, *Rubik's for cryptographers*, Notices Amer. Math. Soc. **60** (2013), 733–739.

[13] A. Rukhin, J. Soto, J. Nechvatal, E. Barker, S. Leigh, M. Levenson, D. Banks, A. Heckert, J. Dray, S. Vo, and others, *A statistical test suite for random and pseudorandom number generators for cryptographic applications*. NIST special publication, (2010).

[14] 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** (1993), 90–110.

[15] J.-P. Tillich and G. Zémor, *Hashing with SL_2*, in CRYPTO 1994, Lecture Notes Comp. Sci. **839** (1994), 40–49.

DEPARTMENT OF MATHEMATICS, THE CITY COLLEGE OF NEW YORK, NEW YORK, NY 10031

E-mail address: shpil@groups.sci.ccny.cuny.edu

QUEENSBOROUGH COMMUNITY COLLEGE AND GRADUATE CENTER, CITY UNIVERSITY OF NEW YORK

E-mail address: bsosnovski@qcc.cuny.edu

TABLE 1. NIST Statistical Suite Results

Number	Statistical test	p-value	Pass rate
1	Frequency	0.924076	99/100
2	Block frequency	0.883171	100/100
3	Cumulative sums 1	0.834308	99/100
4	Cumulative sums 2	0.534146	99/100
5	Runs	0.080519	99/100
6	Longest runs of ones	0.108791	100/100
7	Rank	0.249284	100/100
8	FFT	0.595549	99/100
9..156	Non-overlapping templates (148 tests)	0.507875 (mean)	99/100 (mean)
157	Overlapping templates	0.401199	100/100
158	Universal	0.319084	100/100
159	Approximate entropy	0.924076	99/100
160..167	Random excursions (8 tests)	0.6439595 (mean)	60.8/61 (mean)
167..184	Random excursions (variant - 18 tests)	0 .623016 (mean)	60.9/61 (mean)
185	Serial 1	0.595549	97/100
186	Serial 2	0.983453	100/100
187	Linear complexity	0.202268	100/100