Snapshots of modern mathematics
from Oberwolfach

№ 2/2026

Secure file sharing and Cayley graphs

Eilidh McKemmie

Have you ever wondered how your computer knows it can trust certain downloads but not others? This snapshot describes some security concerns and one algebraic way of dealing with them. We'll see an interesting procedure that uses a very difficult problem in algebra to provide security, and discuss some of the procedure's important properties.

1 File sharing

Imagine you are trying to download a large file from the internet but your enemy, Eve, is determined to ruin your fun. Since the file is large, it is split up into multiple different parts called *packets*. This situation is common with peer-to-peer file sharing. If Eve can tamper with some of the packets she can ruin the whole file, or even insert a virus. You'd like to find this out before you put the packets together and open the corrupted file.

You might use something called a *hash function* which assigns to any possible packet a short digital fingerprint called a *hash*. You'd be sent the hash of each packet through a separate channel. Then you just apply the hash function to the packet and check if it's the same as the claimed hash for that packet. If the actual hash is different from the claimed hash, you know that packet has been tampered with and you discard it before trying to download it again.

We're going to use mathematics to create a hash function which has nice properties for file sharing.

2 Desirable properties

We're going to assume that Eve can mess with the packets but not with the hashes. If the hashes are much smaller than the packets, this means we can use a secure channel which can handle only a small amount of data. Then the small hashes fit through the secure channel, while the big packets do not.

Unfortunately this introduces another problem: there will be far more possible packets than possible hashes, so a hash will not be unique to the packet. That means Eve could find another bogus packet with the correct hash and mess with your file anyway. Therefore you want your hash function to be *pre-image resistant*, which means that it is very difficult for Eve to compute a packet which has a given hash.

There are different ways to put the original file back together from the downloaded packets, and in some situations we can do it by just multiplying together some packets. Depending on what the packages actually are, *multiply* can mean arithmetic multiplication, but also just appending all packages. For example, fountain codes are used in peer-to-peer file sharing to break up the file into packets so that you can reassemble the original file even if you lost a small number of packets [6]. If the hashes of the packets can be multiplied together to get the hash of the file, we can compute hashes a lot more quickly by hashing packages simultaneously, and then multiplying them together. When the product of hashes is the hash of the product of packets, the hash function is called *homomorphic*.

Using a homomorphic hash function means that we only need the hash of the whole file to be sent over our secure channel. We don't have to go through the trouble of constructing the file before checking if it was tampered with. Instead, we can hash each packet as it arrives and then take the product at the end and compare to the expected hash of the file.

So far, we have identified that our objective is to find a function mapping packets to hashes with the following properties:

1. Compression: the hash is much smaller than the packet.
2. Homomorphism: the hash of a product should be the product of hashes.
3. Pre-image resistance: it should be very difficult for Eve to find a packet with a given hash.

One potential issue is that Eve could insert a packet whose hash is just 1, which would not change the product of hashes but would change the file. Therefore we will also ask that our hash is *collision resistant*:

4. Collision resistance: it should be very difficult for Eve to find a packet whose hash is 1. This turns out to be the same as finding two different packets with the same hash.

So we are looking for a hash function which compresses, is a homomorphism, and is pre-image and collision resistant. We will see how to create such a hash function using matrix multiplication.

3 Matrix multiplication

When one number is not enough information, you might use a matrix which is an array of numbers. Here are some examples of matrices whose entries are integers (whole numbers):

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 2 \\ 2 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Matrices are useful tools in many areas of mathematics and the sciences, for further reading on matrices, see Snapshot 5/2019 [3].

In particular, matrices are often used to hold data for data analysis or image compression. For such applications, computers need to be able to efficiently perform operations like multiplication and addition on matrices. We are going to concentrate on multiplying $n \times n$ matrices, which are squares with n rows and n columns. The first example above is a 3×3 matrix and the third is a 2×2 matrix. For a fixed number n, any two of these matrices can be multiplied together, just like numbers can be multiplied. The multiplication of two $n \times n$ matrices A and B works as follows: Each entry of the new matrix is a number that is computed by multiplying the numbers in a row of A with the numbers in the corresponding column of B one by one, matching them by their position, and finally adding up all those products. Computers can do this very efficiently even for large matrices. When you multiply two $n \times n$ matrices, you get another $n \times n$ matrix, so the matrix doesn't grow in size. This will be useful for achieving our first goal of keeping the size of a hash small.

Multiplying matrices has a lot of similarities to multiplying numbers. There is even a matrix I which acts like the number 1 because when you multiply another matrix A by I, you just get A again. The matrix I has the number 1 in every entry of the diagonal from the top left to the bottom right of the matrix and all other entries are equal to 0. This is called the *identity matrix*. Moreover, you can do something like division with some matrices: to "divide" by a matrix A, you find the inverse A^{-1} of A and multiply by that. Note that there are many matrices that do not have an inverse. These matrices are like the number 0 because you cannot divide by them.

Matrix multiplication is more complicated than multiplying numbers. For example, we know that $2 \cdot 3 = 6 = 3 \cdot 2$ but for matrices A and B we usually

have $AB \neq BA$ (for matrix multiplication, we omit the product symbol). For example

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} = \begin{pmatrix} 4 & 5 \\ 2 & 3 \end{pmatrix} \text{ while } \begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 2 \\ 5 & 4 \end{pmatrix}.$$

It turns out this will help us immensely when coming up with our hash function.

One more thing before we create our hash function: we will actually be taking our matrix entries *modulo* a prime p. This means that we will fix a prime number p at the beginning, and then each entry in each matrix will be replaced by its remainder when dividing by p. This will help to keep our hashes small, since there are only p possibilities for each entry of the matrices. For example, if $p = 3$ then 3 will be replaced by 0 and 4 will be replaced by 1, etc. It turns out this does not mess with our multiplication. So the above example taken modulo 3 will be

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 2 & 0 \\ 1 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 2 & 0 \end{pmatrix} \text{ and } \begin{pmatrix} 2 & 0 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 2 \\ 2 & 1 \end{pmatrix}.$$

The matrices that we will use for our hash function come from a special set $\mathrm{SL}_n(p)$ of $n \times n$ matrices modulo a prime number p which have inverses. In our example, the matrices are from $\mathrm{SL}_2(3)$.

4 Cayley hash functions

To create a *Cayley hash function*, you need to pick two matrices and call them A and B, making sure that $AB \neq BA$. As an example, let's use the matrices

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \text{ and } B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

in $\mathrm{SL}_2(3)$, which Gilles Zémor used when he proposed one of the first Cayley hash functions [11].

A packet is a sequence of 0s and 1s of any length. The function then replaces each 1 with a B and each 0 with an A. The hash is then the product of all those matrices in order. We'll call this function h. For example, using Zémor's choice of A and B gives us

$$h(010110001) = ABABBAAAB = \begin{pmatrix} 2 & 0 \\ 1 & 2 \end{pmatrix}.$$

Since we are using $p = 3$, this function always outputs a 2×2 matrix with entries $0, 1$ or 2. In fact, there are only 24 matrices in $\mathrm{SL}_2(3)$, which means

there are only 24 possible hashes. If we label them by numbers from 1 to 24 and use this label as the final hash, our hash is at most two digits long (or 4 bits if we're using 0s and 1s), no matter how large the packet is. We now have a hash function which has our first desired property: the hash is much smaller than the packet.

We also have our second property: the hash is homomorphic. This is because multiplying the packets just means to append them, for example,

$$h(010)h(110) = ABABBA = h(010110).$$

Notice that in general we cannot rearrange the letters in our packet without changing the hash because $AB \neq BA$. This hints that rearranging will not break collision resistance. To understand pre-image and collision resistance more deeply, we need to know why the hash functions are named after Cayley.

5 Cayley graphs

A *Cayley graph* is a visual way to see the pre-image and collision resistance properties of Cayley hash functions. The mathematician Arthur Cayley (1821–1895) was the first to describe these graphs in 1878 and Max Dehn (1878–1952) developed a lot of the basic theory of Cayley graphs in the early 1900s. Cayley graphs were developed to help us understand the mathematics of symmetry, known as *group theory*. The Snapshot 16/2019 [7] discusses some interesting properties of these graphs, some of which we will use later.

We will concentrate only on the applications of Cayley graphs to understand pre-image and collision resistance for Cayley hash functions. These two properties are much harder to achieve and much harder to verify than compression and homomorphism, which is why we'll need this extra tool.

Given two matrices A, B in $\mathrm{SL}_n(p)$ we define the Cayley graph to be a set of dots, one for each matrix in $\mathrm{SL}_n(p)$, with two dots M and N connected by an arrow $M \to N$ in (apple) red if $N = MA$ or in blue if $N = MB$. This means that if you start with the identity matrix I, you can draw the entire graph by "walking" from dot to dot, where each step is just multiplying by A or B on the right. Figure 1 shows the Cayley graph for Zémor's example of

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \text{ and } B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

The lines are labelled by colour with the As (apple) red and the Bs blue. The identity matrix has a red dot so you can find it easily.

You can see that the graph has a lot of repeating patterns but it is not necessarily easy to navigate through. Notice, for example, the triangles that

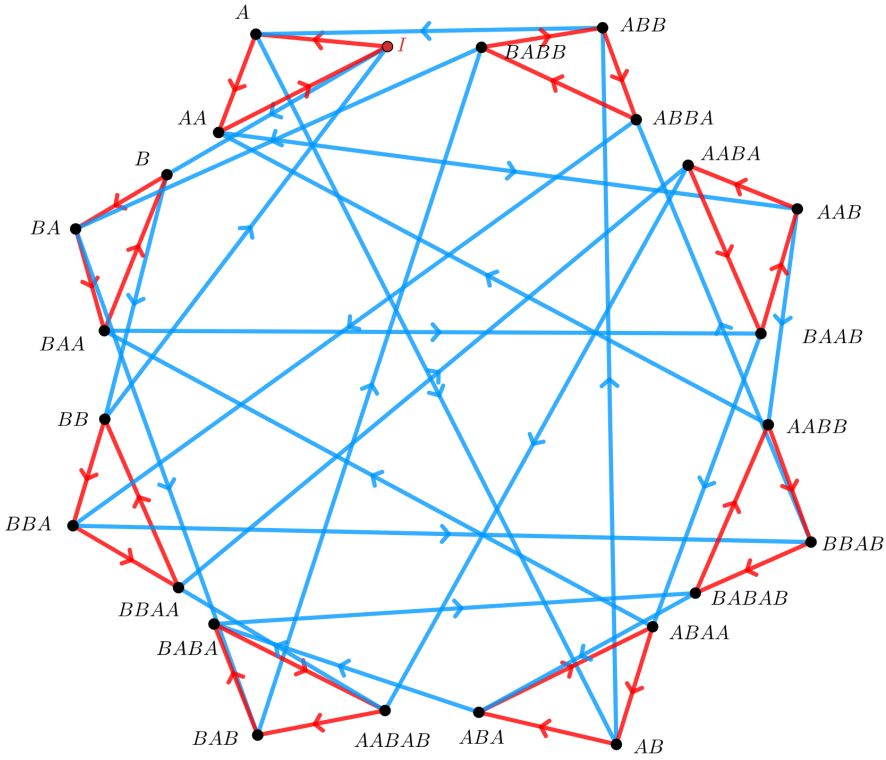

Figure 1: The Cayley graph of $\mathrm{SL}_2(3)$ with Zémor's A and B.

tell us that $AAA = I = BBB$. You can also use the graph to spot facts like $AAB = BAABA$ because the different paths reach the same endpoint.

Now let's think about what pre-image resistance means in terms of the Cayley graph. Pre-image resistance means it is hard to find a packet with a given hash. The hash of a packet is just the endpoint of the packet's path through the Cayley graph. Pre-image resistance can therefore be rewritten as "it is very difficult to find a path in the Cayley graph from the identity I to a given point".

Collision resistance means that it is hard to find a loop in the Cayley graph, because a loop is essentially a packet whose hash is the identity. If you follow the loop in the Cayley graph, you must end in the same matrix you started with. That means, if you multiply exactly those matrices that constitute the loop, you get the identity matrix. Another way to see this is that a loop shows

two different ways of getting to the same place, which means you have two different packets with the same hash.

In our example, the Cayley graph is quite small. You can see that there are loops with just three steps which means the hash is not collision resistant. Additionally, you can walk from the identity to any matrix in at most five steps. Therefore this example is not pre-image resistant: given a matrix M, Eve can simply ask her computer to go through all the possible paths of length five and check if M is at the end of the path. Even if she does not use any tricks to make her work easier, she will only need to check $2^5 = 32$ paths since there are only two possible choices for multiplication in each of the five dots on the path.

The way to get around this problem is to make p very large. You probably want primes large enough to be used in other data security systems like RSA [4], which means primes greater than $2^{2048} \approx 3.2 \times 10^{616}$. Then there will be too many elements in $\mathrm{SL}_n(p)$ and hence far too many paths to check and Eve will need some mathematical tricks to have a chance of breaking pre-image resistance.

6 Choosing the right A and B

Until now we've swept the choice of A and B under the carpet. A Cayley hash function always has the compression and homomorphism properties, but the pre-image and collision resistance of our hash function rely on which A and B we choose.

It is generally considered hard to prove that a given problem is difficult for a computer to solve. This means it is much easier to show that a hash function is *not* pre-image or collision resistant than to show it *is* resistant. Because of this, we will accept good evidence of pre-image or collision resistance in lieu of a proof.

Tillich and Zémor showed that the proposal described above is not collision resistant [10]. Many other suggestions for A and B have also been broken. As far as I know, there are two suggestions which are unbroken [2, 8]. The evidence for pre-image and collision resistance of these proposals comes from girth and freeness properties of the associated Cayley graph.

6.1 Girth

There is a famous conjecture of Babai which says that you need at most around $\log p$ steps to get between any two matrices in the Cayley graph, as long as p is sufficiently large. For $\mathrm{SL}_n(p)$ the conjecture was proven to be true by [5, 1, 9].

To make sure that it is very difficult to break the collision and pre-image resistance of our hash function, we'd need *at least* around $\log p$ steps to get between any two matrices in the Cayley graph. This means that the *girth* of the graph should be as large as possible. Formally, the girth of a Cayley graph is the length of a shortest directed loop in the graph and hence exactly the parameter we identified as important for collision resistance in the previous section.

When we select A and B, we would like their Cayley graph to have large girth. Note that this happens quite often, for example for many families of expander graphs which you can read about in the Snapshot 16/2019 [7].

6.2 Freeness

The reason we are taking remainders after dividing by p for the entries of the matrix is to make sure the hashes stay small. What if you could forget about the remainders and just work with the integers normally? This is called *lifting to the integers*. If you get a violation of collision or pre-image resistance when you lift your A and B to the integers then you've found a violation of collision or pre-image resistance which works for *every* prime p, completely breaking the hash function.

Therefore, we'd like there to be no possible violations of collision or pre-image resistance after A and B are lifted to the integers. If this is the case, we say that A and B *generate a free structure*.

The choices of A and B in [2, 8] generate free structures and their Cayley graphs have large girth, which provides good evidence that the Cayley hash functions are secure.

7 Unanswered questions

In this snapshot, we have seen that Cayley hash functions are promising candidates to provide security in file sharing. We have seen how the choice of matrices A and B can affect the underlying Cayley graph which controls the pre-image and collision resistance properties of our hash function. We understood that choosing secure A and B is difficult, and that mathematicians have some good evidence for certain choices to be safe based on girth and freeness.

The question which remains unanswered is whether these girth and freeness properties are enough to guarantee security. To show that the answer is no, mathematicians would have to come up with an example of A and B with nice girth and freeness properties but without pre-image and collision resistance. To show that the answer is yes would be even harder because researchers will have to consider *every* possible A and B and show that *every* possible algorithm to find pre-images or collisions fails.

Image credits

Figure 1 Drawn using GeoGebra.

References

[1] E. Breuillard, B. Green, and T. Tao, *Approximate subgroups of linear groups*, Geometric and Functional Analysis **21** (2011), no. 4, 774–819.

[2] L. Bromberg, V. Shpilrain, and A. Vdovina, *Navigating in the Cayley graph of $SL_2(\mathbb{F}_p)$ and applications to hashing*, Semigroup Forum **94** (2015), no. 2, 314–324.

[3] A. Detinko, D. Flannery, and A. Hulpke, *Algebra, matrices, and computers*, Snapshots of modern mathematics from Oberwolfach (2019), no. 05, https://doi.org/10.14760/SNAP-2019-005-EN.

[4] M. Gardner, *Mathematical games*, Scientific American **237** (1977), no. 2, 120–125.

[5] H. A. Helfgott, *Growth and generation in $SL_2(\mathbb{Z}/p\mathbb{Z})$*, Annals of Mathematics **167** (2008), no. 2, 601–623.

[6] N. Johnson, *Damn cool algorithms: Fountain codes*, 2012, http://blog.notdot.net/2012/01/Damn-Cool-Algorithms-Fountain-Codes, visited on 2025-28-11.

[7] A. Khukhro, *Expander graphs and where to find them*, Snapshots of modern mathematics from Oberwolfach (2019), no. 16, https://doi.org/10.14760/SNAP-2019-016-EN.

[8] C. Le Coz, C. Battarbee, R. Flores, T. Koberda, and D. Kahrobaei, *Post-quantum hash functions using* $\mathrm{SL}_n(\mathbb{F}_p)$, Advances in Mathematics of Communications **19** (2025), no. 3, 996–1009.

[9] L. Pyber and E. Szabó, *Growth in finite simple groups of lie type*, Journal of the American Mathematical Society **29** (2014), no. 1, 95–146.

[10] J. Tillich and G. Zémor, *Group-theoretic hash functions*, Algebraic Coding, Springer, 1994, pp. 90–110.

[11] G. Zémor, *Hash functions and Cayley graphs*, Designs, Codes and Cryptography **4** (1994), no. 3, 381–394.

Eilidh McKemmie *is an assistant professor of mathematics at Kean University.*

Mathematical subjects
Algebra and Number Theory

Connections to other fields
Computer Science

DOI
10.14760/SNAP-2026-002-EN

Snapshots of modern mathematics from Oberwolfach provide exciting insights into current mathematical research. They are written by participants in the scientific program of the Mathematisches Forschungsinstitut Oberwolfach (MFO). The snapshot project is designed to promote the understanding and appreciation of modern mathematics and mathematical research in the interested public worldwide. All snapshots are published in cooperation with the IMAGINARY platform and can be found on www.imaginary.org/snapshots and on www.mfo.de/snapshots.

ISSN 2626-1995

Mathematisches Forschungsinstitut
Oberwolfach gGmbH
Schwarzwaldstr. 9–11
77709 Oberwolfach
Germany

Director
Gerhard Huisken

Mathematisches
Forschungsinstitut
Oberwolfach

IMAGINARY
open mathematics