Autoproduct Functions and Applications

Preliminary definitions:

```
- matrices are indicated by upper case letters e.g. A
- vectors  are indicated by bold letters e.g. x
- scalars  are indicated by standard letters e.g. x
- `Ax` indicates the matrix product of A and x
- `s * A` indicates the scaling of the coefficients of the matrix A by the scalar va
- `y . x` indicates the dot product of the vectors y and x
- x_0, x_i indicates the i-th element of the vector x
- `xy` and `x_i y_i` indicate the scalar multiplication of x and y and x_i, y_i resp
- `x + y` indicates the scalar addition of x and y
- `A + B` indicates the entry-wise scalar addition of the entries of A and B
- F(A, x) indicates the application of the function F with the arguments A, x
- s^2 indicates the exponentiation of the scalar s by the value 2 (i.e. squaring)
```

Autoproduct functions

Consider the following transformation that maps a matrix and a vector to a scalar:

```
Ax . x
```

the dot product between a vector x and a transformation of itself `Ax`.
This type of transformation is referred to as an "autoproduct".

Algebraic structure

Autoproduct functions have some usable algebraic structure.

Viewing the transformation as the dot product between a vector x and some vector y = Ax:

```
Ax . x = y . x
        = x_0 y_0 + x_1 y_1 + ...
        = ax + by + ...
```

```
Bx . x = z . x
        = x_0 z_0 + x_1 z_1 + ...
        = dx + ey + ...

Ax . x + Bx . x = (y . x) + (z . x)
                 = (y + z) . x      # dot product is distributive
                 = (A + B)x . x     # matrix product distributes over matri
```

Rewriting using function notation F(A, x) -> Ax . x:

```
F(A, x) + F(B, x) = F(A + B, x)
```

Scalar multiplication distributes over the left matrix argument:

```
s * F(A, x) = F(s * A, x)
```

Scalar multiplication distributes a square root of the scalar over the right vector argument:

```
s^2 * F(A, x) = F(A, s * x)
```

This is a consequence of s * x being multiplied with a transformation of itself:

```
A(s * x) . (s * x)
s * Ax . (s * x)
(s * y) . (s * x)
sa, sb, ... . sx, sy, ...
ssax + ssby + ...
```

Polynomial defined by the matrix

Section is still under construction.

First look at permutation matrices.

Then random matrices with coefficients in (-1, 0, 1).

Then random matrices with random coefficients.

Types of autoproduct functions

- Permutation matrices
- Signed permutation matrices
- Generalized permutation matrices
- Random matrices with coefficients in (-1, 0, 1)
- Random matrices with small random coefficients
- Random matrices
- Higher order variants
    - Product of subset sums
    - Sum of subset products
- Public fixed matrix, private vector
- Private matrix, public fixed vector
- Tensor autoproduct (uses an array of matrices instead of 1 matrix)
    - Further generalizations of either argument to N dimensions

Basic attack on permutation matrix variant

Put all possible pairs of coefficients from the vector (quadratic monomials) into a matrix for LLL.

LLL matrix size: O(n^2), where n is the dimension of the target secret permutation matrix.

Cost of LLL in terms of dimension:

```
O(n^4) (or O(n^5)?)
O(n^(4 * 2)) (or O(n^(5 * 2))?)
O(n^8) (or O(n^10))
```

A quadratic advantage.

If n=2^16, then 2^16^8 = 2^128 and the attack will cost too much to compute.
But such a matrix is still large.

Signed permutation matrix variant

Same attack as before, with more pairs because coefficients come from (-1, 0, 1) instead of (0, 1)

LLL matrix size:

```
O(2n^2)
```

Cost of LLL:

```
O((2n^2)^4) = O(2^8 n^8)
```

If n=2^15, then 2^8 * (2^15^8) = 2^8 * 2^120 = 2^128

Still a quadratic advantage, with a scaling factor.
Saved 32,768 * log2(coefficient size) bits of space.

But this is still large.

Random matrices with coefficients in (-1, 0, 1)

Each vector in the matrix is independent. This is distinct from the permutation matrix based variants. Previously a process of elimination limited the advantage to being quadratic.

There are 3^n vectors of dimension n with coefficients in (-1, 0, 1).

LLL matrix size:

```
O(3^(n^2))
```

Cost of LLL:

```
O(3^(n^2)^4) = O(3^(4(n^2)))
```

If n=2^3=8, then

```
3^(4(8^2)) = 3^(4 \* 64) = 3^(256) = 2^(1.58 \* 256)
```

This attack is clearly inefficient here.

At this point, the matrix only has 64 coefficients in 8 rows/columns.
Guessing a good number of rows/columns is feasible.
The dimension should be chosen large enough to block these attacks too.

```
- How many rows/columns are needed to help another attack?
```

Randomly generated rows have a probability of colliding.
Given some row vector v of dimension n, what is the probability that a uniformly random row is identical to v?

```
- There are 3^n possible row vectors
- They are sampled uniformly
- Only one of them is equal to v
- 1/(3^n)
```

Each redundant row removes 3^n combinations from the attack matrix.

- If all rows are the same ...
- If all rows are mostly the same ...
- ...
- If all rows are distinct ...
- What about linearly dependent rows instead of equal rows?

In order to block this completely, make rows long enough to to provide an acceptable probability of collisions:

```
3^n >= 2^k
```

Where k is the number of bits of security expected. If k=256:

```
3^n ~= 2^(n1.58) >= 2^256
3^162 ~= 2^(162 * 1.58) >= 2^256
```

Autoproduct-based homomorphic hashing

A "weak" hash function works as a hash for random inputs. The tensor autoproduct can be used to instantiate this construction.

```
F(M, k) -> Mk . k -> t
F(M1, k) + F(M2, k) = F(M1 + M2, k)
```

where:

```
- M is a tensor (an array of matrices) whose coefficients are the bits of the messag
    - "message" in this case means some randomized information, e.g. ciphertext.
- k is a fixed vector
- t is the output vector (or possibly a scalar if M is only a matrix)
```

By keeping k secret, the construction becomes a keyed hash function.

- Can use it as a MAC

Applicability

Ciphertext that is proven to be indistinguishable from random is suitable for use as an input.

- Can hash ciphertexts from partially homomorphic schemes without breaking the structure
- Can create MACs for ciphertexts from partially homomorphic schemes without breaking the

structure

It can be used to verify that a sum contains only elements from a prescribed set of values.