Proceedings of the Federated Conference on Computer Science and Information Systems pp. 221–225
DOI: 10.15439/2019F277
ISSN 2300-5963 ACSIS, Vol. 18

Accelerating Multivariate Cryptography with Constructive Affine Stream Transformations

Michael Carenzo
Rochester Institute of Technology
1 Lomb Memorial Dr,
Rochester, NY 14623
Email: mec2487@rit.edu

Monika Polak
Rochester Institute of Technology
1 Lomb Memorial Dr,
Rochester, NY 14623
Email: mkpvcs@rit.edu

***Abstract*—On December 20th, 2016, the National Institute of Standards and Technology (NIST) formally initiated a competition to solicit, evaluate, and standardize one or more quantum-resistant cryptographic algorithms. Among the current candidates is a cryptographic primitive which has shown much promise in the post-quantum age, Multivariate Cryptography. These schemes compose two affine bijections S and T with a system of multivariate polynomials. However, this composition of S and T becomes costly as the data encrypted grows in size. Here we present Constructive Affine Stream (CAS) Transformations, a set of algorithms which enable specialized, large-scale, affine transformations in $O(n)$ space and $O(n \log n)$ time, without compromising security. The goal of this paper is to address the practical problems related to affine transformations common among almost all multivariate cryptographic schemes.**

I. Introduction

MULTIVARIATE Cryptography is a, post-quantum, cryptographic primitive based on the difficulty of solving systems of multivariate equations over a finite field [1]. At their core, multivariate schemes define a set of (usually quadratic) polynomials:

$$\begin{pmatrix} p_1(w_1, ..., w_n) \\ ... \\ p_m(w_1, ..., w_n) \end{pmatrix}$$

where all coefficients and variables are in $\mathbb{F}_q$, a field with q elements. Given a plaintext message: $(x_1, ..., x_n) \in \mathbb{F}_q^n$ the ciphertext is computed by evaluating:

$$\mathcal{P}(x_1, ..., x_n) = \begin{pmatrix} p_1(x_1, ..., x_n) \\ ... \\ p_m(x_1, ..., x_n) \end{pmatrix} = \begin{pmatrix} c_1 \\ ... \\ c_m \end{pmatrix}$$

To decrypt the ciphertext $(c_1, ..., c_m)$, one must hold the *secret key* used to generate the polynomials in $\mathcal{P}$ in order to invert $\mathcal{P}$.

$$\mathcal{P}^{-1}(c_1, ..., c_m) = (x_1, ..., x_n)$$

Inverting $\mathcal{P}$ without the secret key is equivalent to solving a system of multivariate equations over a finite field, known formally as the $\mathcal{MQ}$-Problem, and is proven to be NP-Hard. However, modern constructions of these schemes rarely use a set of multivariate polynomials on their own. Modern constructions almost always compose the set of polynomials with two invertible affine maps S and T [2]. So, in reality:

$$\mathcal{P} = T \circ Q \circ S : \mathbb{F}_q^n \to \mathbb{F}_q^m$$

Where Q (also known as the *central* or *core map*) is the set of multivariate polynomials and S and T are (sometimes linear) affine maps of full-rank. While many papers focus on the design of the central map, few describe how to effectively generate and compose S and T, despite the importance of this operation [3], [4], [5], [6], [7]. As the plaintext grows in size, so too do S and T. In fact, S and T grow so fast that their composition becomes intractable very quickly.

This poses a significant hurtle for symmetric applications of multivariate cryptography. Consider encrypting a 1kB, 500kB, and 1MB file. Because S and T are $n \times n$ matrices where n is the size of the plaintext, these files require matrices 1MB (1kB^2), 250GB (500kB^2), and 1TB (1MB^2) in size. This rapid inflation of S and T restricts these schemes from (reasonably) encrypting anything larger than ~1kB (without chaining).

In this paper we present Constructive Affine Stream (CAS) Transformations, a set of algorithms capable of efficiently generating and multiplying S and T by any arbitrary vector. At the same time, these transformations preserve the post-quantum security of multivariate ciphers. We begin by presenting the theory behind these transformations (Sec. II), followed by a general implementation (Sec. III). We then analyze the asymptotics of the aforementioned implementation (Sec. IV) and conclude by evaluating how these transformations impact the security of multivariate ciphers (Sec. V).

II. Constructive Affine Stream Transformations

Instead of generating, storing, and operating on a matrix outright, a Constructive Affine Stream (CAS) deterministically generates (via some seed derived from the private key) a stream of integers that represent an affine matrix of full-rank. These streams can then be used to transform a vector progressively, in the same way a normal matrix-vector multiplication would, without ever having to store an actual matrix. Furthermore, a stream (if configured with the same seed) can be switched to generate the inverse of a given matrix so that a previous transformation can be undone.

A. The Structure of an Affine Stream

Constructive Affine Streams (CAS) leverage a basic property of matrix-vector multiplication; each (matrix) row is dotted with the vector term, one at a time, independently of

the other rows. In effect, this property allows the values of each row to be randomly generated over the course of its dot product with the vector term and then "thrown away." This randomly generated sequence of values is what an affine stream is composed of and allows a vector to be transformed without having to store a matrix.

However, randomly generating matrix rows doesn't guarantee that the resulting matrix is invertible. Furthermore, even if the matrix was invertible, the values of the rows are generated as needed and never stored, thus each matrix value is "blind" to the values adjacent to it. This limitation prevents a matrix's inverse from being computed using conventional techniques. In order to solve this problem, matrices generated by a CAS maintain a specific structure.

A CAS generated matrix:

1) Takes the form of an upper or lower triangular matrix;
2) With non-zero values on the main-diagonal; and
3) For every row/column pair which intersect on the main-diagonal, only one of the two (row or column) can contain non-zero values.

While conditions 1 and 2 ensure that the matrix stream produced is always invertible, condition 3 guarantees that the matrix stream is invertible, one row at a time, using only values on the main-diagonal. While the result of condition 3 may not seem intuitive at first, consider inverting the matrix in Fig. 1 via Gauss-Jordan elimination.

A row that only contains zeros (excluding the main-diagonal) can only eliminate values down the column which intersects it on the main-diagonal. For instance, in Fig. 1, r_1 will only be used to eliminate values in c_1. A row that contains multiple non-zero values has nothing to eliminate in the column which intersects it on the main-diagonal because, by the definition above, that column will always contain zeros. Returning to Fig. 1, r_3 will never need to eliminate anything down c_3 because it is guaranteed to be "zero-valued" by definition. However, r_3's non-zero values 5 and 4 will be eliminated by rows r_0 and r_1 respectively.

Described more generally, the only rows which perform elimination are the ones which contain a single non-zero value: the diagonal-component. Moreover, the only columns which contain non-zero values are the columns which intersect one of the aforementioned rows on its diagonal-component. Consequently, each column can be eliminated independently, without altering adjacent columns.

Because every column can be eliminated on its own, it follows that every column can be inverted on its own. This result is what enables on-the-fly, row-by-row, CAS inversion. The only information needed beforehand are the values of the main-diagonal. With these values, each row can be inverted, one at a time, by mapping its diagonal-component to its multiplicative inverse and all other values (off the main-diagonal) via:

$$-\alpha_{i,j} \times \alpha_{j,j}^{-1} \times \alpha_{i,i}^{-1}$$

where $\alpha_{i,j}$ is the matrix value at the ith column and jth row.

Form	5×5 CAS
Matrix Representation	$\begin{array}{c} \\ r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{array} \begin{array}{c} \begin{array}{ccccc} c_0 & c_1 & c_2 & c_3 & c_4 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 8 & 0 & 1 & 0 & 0 \\ 5 & 4 & 0 & 1 & 0 \\ 0 & 7 & 0 & 0 & 1 \end{bmatrix} \end{array}$
Stream	[1,0,8,5,0,1,0,4,7,1,1,1]

Fig. 1. An Example CAS Stream

The derivation of this equation is fairly straight forward. Given the following augmented matrix:

$$\left[\begin{array}{ccc|ccc} \alpha_{i,i} & \cdots & 0 & 1 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{i,j} & \cdots & \alpha_{j,j} & 0 & \cdots & 1 \end{array}\right]$$

we can solve for the inverse matrix value at position $\alpha_{i,j}$, via Gauss-Jordan elimination. Here, the rows at i and j are normalized:

$$\left[\begin{array}{ccc|ccc} 1 & \cdots & 0 & \alpha_{i,i}^{-1} & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{i,j} \times \alpha_{j,j}^{-1} & \cdots & 1 & 0 & \cdots & \alpha_{j,j}^{-1} \end{array}\right]$$

then the row at i (multiplied by $\alpha_{i,j} \times \alpha_{j,j}^{-1}$) is subtracted from the row at j:

$$\left[\begin{array}{ccc|ccc} 1 & \cdots & 0 & \alpha_{i,i}^{-1} & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 1 & \underline{-\alpha_{i,j} \times \alpha_{j,j}^{-1} \times \alpha_{i,i}^{-1}} & \cdots & \alpha_{j,j}^{-1} \end{array}\right]$$

Note that in Fig. 1, the main-diagonal only contains ones. This matrix configuration is known formally as a "Semi-Byte" CAS and is one of the three main stream types. A Semi-Byte CAS is trivial to invert because the main-diagonal doesn't need to be generated beforehand (as it is already known) and each value off of the main-diagonal is mapped via:

$$-\alpha_{i,j} \times 1^{-1} \times 1^{-1} = \underline{-\alpha_{i,j}}$$

B. Performing Constructive Affine Stream Transformations

Notice that the affine stream in Fig. 1 is not a consecutive series of matrix rows. These streams take advantage of the aforementioned matrix constraints to generate only the values necessary for vector transformation. Structurally, affine streams are organized by column and do not contain the zeroed half of the triangular matrix. Furthermore, they do not contain the zeros of zero-valued columns, only their diagonal component. For example, in Fig. 1, 8 was generated in r_2, therefore c_2 must be a zero-valued column. The only value in the stream from c_2 is its diagonal component: 1.

With these streams, a CAS Transformation can be applied to a vector (*vector*) by iterating over a CAS stream and, for each stream value v, we apply:

$$\textit{transform}[r_v] \leftarrow \textit{transform}[r_v] + (\textit{vector}[c_v] \times v)$$

Where *transform* is an empty vector which stores the transformation and r_v and c_v are the row/column matrix coordinates of v. (e.g. In Fig. 1, 8 has the (r_v, c_v) coordinates `(2,0)`.)

III. Implementation

There are three (main) types of CAS: Binary, Semi-Byte and Byte. Each type requires methods for deterministically generating random bits and random non-zero numbers. The bits determine where non-zero values are placed in a matrix while the numbers determine what the values are. If the sequence of bits and numbers can't be regenerated, a transformation can't be inverted. For the sake of our experiments, we leveraged Trivium ([8]) for our bit generator and a simple linear congruential generator (LCG) for our number generator.

Note that in the implementation that follows: *rand* refers to some Pseudo-Random Number Generator (PRNG) initialized with a password which seeds the bit and number generator. (This password can be the same one used for multivariate encryption.) *lowerTriangular* is a Boolean indicating whether the generated CAS matrix is upper or lower triangular. Lastly, the operators $*$ and $+$ refer to group multiplication and addition specific to the chosen finite field.

A. Semi-Byte CAS Transformations

A Semi-Byte CAS generates an affine stream composed of 1's down the main-diagonal and 8-bit values everywhere else. Despite its name, matrix values aren't limited to 8-bits and should operate in whatever finite field is selected for encryption. (e.g. $GF(2^8) \rightarrow$ 8-bit matrix values, $GF(2^{16}) \rightarrow$ 16-bit matrix values) To save space, the implementation described below performs the vector transformation in-place. However, it could easily be modified to store the values in a new vector.

Algorithm 1 Semi-Byte CAS Transformation of a *vect*

```
1:  emptyColumns ← [0] * vect.length
2:  for i = 0 to vect.length do
3:      if emptyColumns[i] == 0 then
4:          for j = 1 to vect.length − i do
5:              if rand.getBit() == 1 then
6:                  scalar ← rand.getByte()
7:                  if lowerTriangular then
8:                      vect[i + j] ← vect[i + j] + (vect[i] ∗ scalar)
9:                  else
10:                     vect[i] ← vect[i] + (vect[i + j] ∗ scalar)
11:                 end if
12:                 emptyColumns[i + j] ← 1
13:             end if
14:         end for
15:     end if
16: end for
```

In order to perform an inverse transformation (relative to a given seed) this implementation would be altered as follows:

- `Line 8:` *vect*[i + j] ← *vect*[i + j] − (*vect*[i] ∗ *scalar*)
- `Line 10:` *vect*[i] ← *vect*[i] − (*vect*[i + j] ∗ *scalar*)

B. Binary CAS Transformations

A Binary CAS generates an affine stream composed of only 0's and 1's. Its implementation is practically identical to that of a Semi-Byte CAS Transformation. However, in the case of Binary CAS Transformations, *scalar* is always equal to 1.

C. Byte CAS Transformations

A Byte CAS generates an affine stream composed of 8-bit values. (Again, bear in mind that values aren't necessarily fixed to 8-bits and, in reality, are bound by the chosen finite field.) These transformations cannot be done in-place and are the costliest in-terms of space-complexity. Implementation details regarding this transformation type can be found at [9].

IV. Asymptotic Analysis & Performance

Compared to typical matrix-vector multiplication, CAS Transforms are quite efficient. In terms of space complexity, both Binary and Semi-byte CAS Transforms require only $\Theta(n)$ space to store the columns flagged as zero-valued (*emptyColumns*). Byte CAS Transforms require $2n$ space for transformations and $3n$ for inversions, giving both operations a lower-bound of $\Omega(n)$.

Computing time-complexity is slightly more complicated due to the probability involved in CAS generation. While normal matrix-vector multiplication is an n^2 operation (for square matrices), CAS Transformations effectively "skip" zero-valued columns as they only operate on their diagonal component. Because these columns are skipped, we can compute the average upper-bound of a CAS Transform by multiplying the height of the columns by the average number of non-zero columns. However, this requires a function which can approximate the average number of non-zero columns a CAS will generate given a matrix size.

While there are several approaches that could be used to derive this function, we chose a statistical approach as it seemed to yield the best estimates. This approach involved randomly generating 5000 (Binary) CAS matrices at each size ranging from 1×1 to 3000×3000 and computing the average number of non-zero columns at each size. Each matrix was generated using its own instantiation of Trivium initialized with the first 160-bits of a SHA-256 hash derived from a (randomly generated) alphanumeric password. Plotting these results, it appears that the number of non-zero columns grows in a logarithmic fashion. Indeed, once we fitted a logarithmic curve to the data we found that it fit perfectly (Fig. 2a).

This would seem to indicate that the runtime of a CAS Transformation is on average $O(n \log n)$. To verify this result, we plotted the average length of a CAS stream. Using the same parameters, (5000 trials over 3000 sizes) we found that the average length of a CAS stream perfectly fits an $n \log n$ curve (Fig. 2b). Note, this run-time applies to all CAS types because all types generate their matrix structure using the same bit generation algorithm (in our case, via Trivium).

With the proper implementation, these transformations have the potential to reach impressive speeds and are limited primarily by their chosen bit and number generation algorithms.

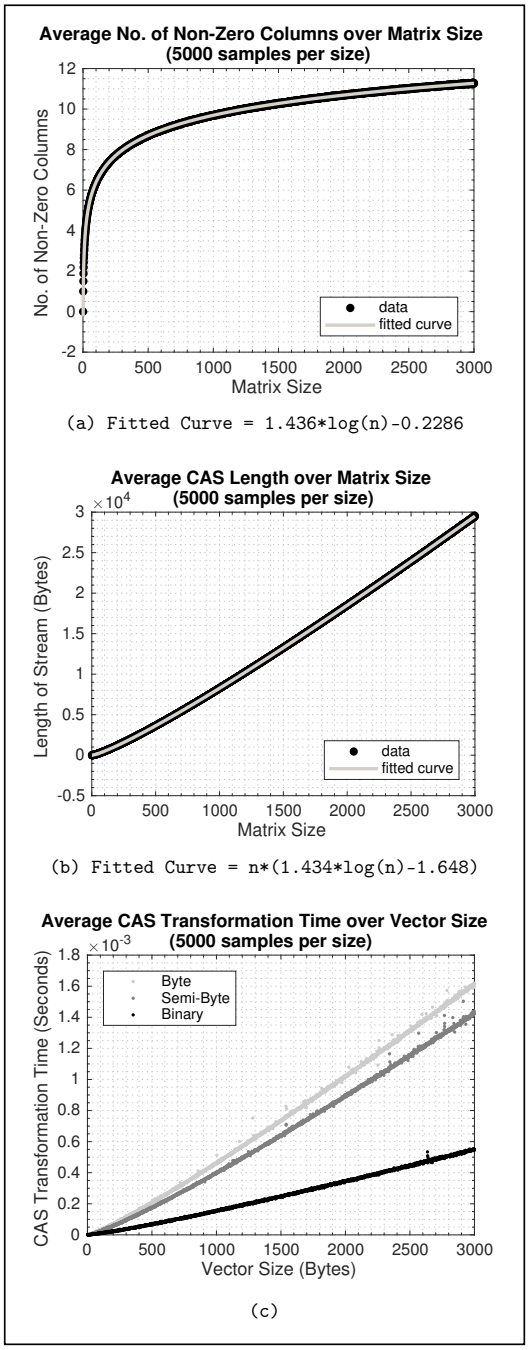

Fig. 2. Runtime Statistics & Time Plots

Our naïve C implementation (run on a computer with an Intel Core i5-3570 processor running at 3.40GHz using 16GB of RAM, with Windows 10 Pro) achieved respectable speeds on its own (Fig. 2c). However, it could be made to operate even faster with an optimized Trivium implementation. In theory a CAS stream could even be cached and applied over blocks. Observe that our speed tests further support our time-complexity analysis as the speed plots reflect our time-complexity plot.

V. Security

In practice, one CAS Transformation alone isn't enough to sufficiently "mix-up" an input vector. In fact, any triangular matrix on its own isn't enough. The use of single triangular matrices can even break certain multivariate schemes. For example, the multivariate scheme based on the family of expander graphs $D(n, q)$ is rendered totally insecure when S and T take the form of lower-triangular matrices. (For more details about this family of graphs see [10].) To illustrate this insecurity, consider Example 5.1.

Example 5.1 (The "Poor Mixing" Vulnerability):

$$Plaintext = [x_1, x_2, x_3, x_4]$$
$$Password = [2, 1, 10, 5]$$
$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}, \quad S = T^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

(Note: Often $S = T^{-1}$ but it is not a general rule.)

The resulting polynomials produced using S, T, and the polynomials generated from $D(n = 4, q)$ graphs (traversed with $Password$) are:

$$p_1(x_1) = x_1 + 18$$
$$p_2(x_1, x_2) = -18x_1 + x_2 - 201$$
$$p_3(x_1, x_2, x_3) = -18x_1^2 - 201x_1 - 18x_2 + x_3 - 513$$
$$p_4(x_1, x_2, x_3, x_4) = 36x_1^2 + 621x_1 + 18x_2 + x_4 + 3261$$

For the sake of simplicity, these polynomials are not bound to any specific finite field $GF(q)$. (All work is shown at [9].)

Notice that each subsequent polynomial introduces one new variable (x_i). So, given the piece of ciphertext c_1 produced by $p_1(x_1)$, it would be trivial to compute the plaintext piece x_1 by solving $x_1 = c_1 - 18$. Then, given x_1, we could solve for x_2 in $p_2(x_1, x_2)$ by plugging in x_1 and c_2. This process can be repeated until all x_i's have been solved for and the plaintext message is revealed (without use of the private key).

To eliminate this vulnerability, a lower and upper triangular CAS Transformation can be combined to form a "Square CAS Transformation." This is simply achieved by applying an upper and lower triangular CAS Transformation to a vector, in any order. Because matrix multiplication is associative, this is equivalent to multiplying a vector by the product of an upper and lower triangular CAS matrix. In fact, simulations have shown that a combined upper and lower CAS Transformation correspond to multiplication by a matrix A, such that for all $i \neq j$ $P(a_{i,j} = 0) \approx 0.65$ and $i = j$ $P(a_{i,j} = 0) \approx 0.35$ [9].

Illustrated below are the corrected polynomials which leverage an upper and lower triangular matrix for both S and T:

$$p_1(x_1, x_2, x_3, x_4) = -54x_1^2 - 216x_1x_2 - 108x_1x_3 - \cdots$$
$$p_2(x_1, x_2, x_3, x_4) = 18x_1^2 + 72x_1x_2 + 36x_1x_3 + \cdots$$
$$p_3(x_1, x_2, x_3, x_4) = -18x_1^2 - 72x_1x_2 - 36x_1x_3 - \cdots$$
$$p_4(x_1, x_2, x_3, x_4) = 36x_1^2 + 144x_1x_2 + 72x_1x_3 + \cdots$$

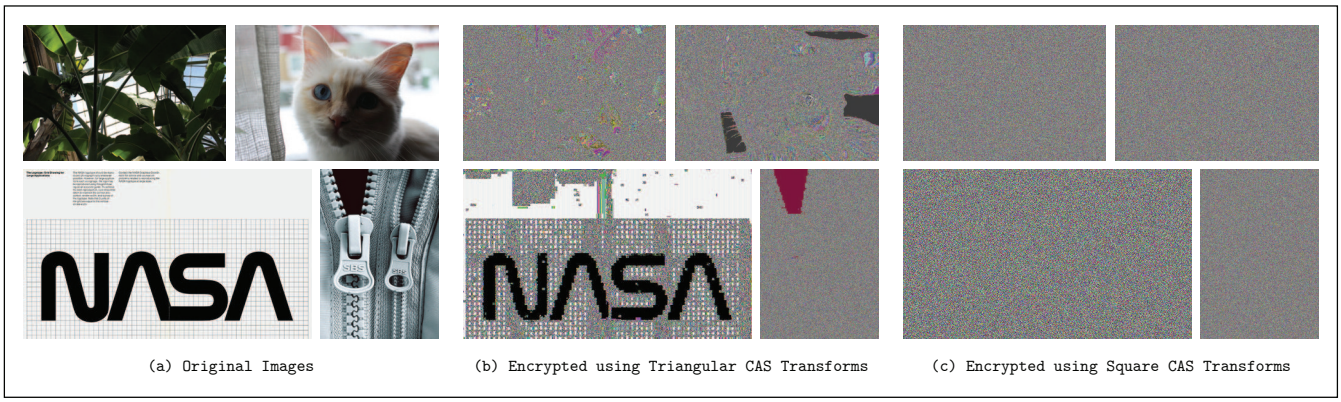

Fig. 3. Image Encryption Tests

A Square CAS Transform can easily be inverted by applying the inverse of its lower and upper triangular components in reverse order. That is, given a vector σ, its transformation σ', and a triangular CAS Transform $T(x)$:

$$T_{lower}(T_{upper}(\sigma)) = \sigma', \quad T^{-1}_{upper}(T^{-1}_{lower}(\sigma')) = \sigma$$

We can visualize this "poor mixing" vulnerability by encrypting image data. In Fig. 3, the images in `(a)` are composed with an upper-triangular (Semi-Byte) CAS T, passed through a set of polynomials (from $D(n, q)$), and then composed with another upper-triangular CAS S to produce the images in `(b)`. Clearly, a large amount of information is exposed. However, if we utilize a square (Semi-Byte) CAS for both S and T we produce the images in `(c)`. (Additional tests with alternate configurations can be found at [9].)

While in theory more than two CAS Transformations could be applied to a single vector, Square CAS Transforms on their own have shown to be sufficient.

To further validate the security of Square CAS Transforms, we analyzed the average order of each (square) CAS type. This statistic is of particular interest as there is theoretical evidence which suggests that large order matrices provide better security [11]. (The order of a matrix $M_{n\times n}$ over a finite field is the smallest positive integer k such that $M^k = I_n$.) At present, the resources required to compute k over large matrices has restricted our ability to collect large samples of data. However, our initial tests indicate that the order of a Binary matrix rarely exceeds $k = 120$. In contrast, Semi-Byte and Byte matrices have both exhibited k's in the millions and even billions [9]. This suggests what one might already expect; Semi-Byte and Byte Transforms are considerably stronger than Binary Transforms, making them the preferred choice for most implementations.

VI. Conclusion

As quantum computers march closer towards their full realization we must be prepared with a new set of cryptographic primitives which remain secure in the post-quantum age. Multivariate Cryptography is one of the handful of primitives which has shown real promise but still requires more research and development before it can be realistically implemented. In this paper we formalized the notion of CAS Transformations, a set of algorithms capable of performing specialized, large-scale, affine transformations in $O(n)$ space-complexity and $O(n \log n)$ time-complexity. While there is still more to be learned about these transformations, they have demonstrated real potential; capable of increasing the speed and scale at which Multivariate Cryptography can be applied.

References

[1] D. J. Bernstein, *Introduction to post-quantum cryptography*. Springer, Berlin, Heidelberg, 2009. [Online]. Available: doi.org/10.1007/978-3-540-88702-7_1

[2] J. Ding and B. Y. Yang, *Multivariate Public Key Cryptography*. Springer, Berlin, Heidelberg, 2009. [Online]. Available: doi.org/10.1007/978-3-540-88702-7_6

[3] J. Ding, M. S. Chen, A. Petzoldt, D. Schmidt, and B. Y. Yang. (2019) Rainbow digital signature algorithm, nist post-quantum cryptography project, round 2 submissions. [Online]. Available: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions

[4] K. Sakumoto, T. Shirai, and H. Hiwatari, *Public-Key Identification Schemes Based on Multivariate Quadratic Polynomials*. Springer, Berlin, Heidelberg, 2011. [Online]. Available: doi.org/10.1007/978-3-642-22792-9_40

[5] C. Tao, H. Xiang, A. Petzoldt, and J. Ding, "Simple matrix - a multivariate public key cryptosystem (mpkc) for encryption," *Finite Fields and Their Applications*, 2015. doi: 10.1016/j.ffa.2015.06.001. [Online]. Available: doi.org/10.1016/j.ffa.2015.06.001

[6] M. Polak, U. Romańczuk, V. Ustimenko, and A. Wróblewska, "On the applications of extremal graph theory to coding theory and cryptography," *Electronic Notes in Discrete Mathematics*, 2013. doi: 10.1016/j.endm.2013.07.05. [Online]. Available: doi.org/10.1016/j.endm.2013.07.051

[7] V. Ustimenko, U. Romańczuk-Polubiec, A. Wróblewska, M. Polak, and E. Zhupa, "On the constructions of new symmetric ciphers based on nonbijective multivariate maps of prescribed degree," *Security and Communication Networks*, 2019. doi: 10.1155/2019/2137561. [Online]. Available: doi.org/10.1155/2019/2137561

[8] C. D. Cannière and B. Preneel, *Trivium*. Springer, Berlin, Heidelberg, 2008. [Online]. Available: doi.org/10.1007/978-3-540-68351-3_18

[9] M. Carenzo. (2019) Study of multivariate cryptography, rit independent study website. [Online]. Available: https://www.cs.rit.edu/~mec2487

[10] F. Lazebnik, V. A. Ustimenko, and A. J. Woldar, "A new series of dense graphs of high girth," *Bull. Amer. Math. Soc.*, 1995. doi: 10.1090/S0273-0979-1995-00569-0. [Online]. Available: doi.org/10.1090/S0273-0979-1995-00569-0

[11] V. Ustimenko, "On the families of stable multivariate transformations of large order and their cryptographical applications," *Tatra Mountains Mathematical Publications*, 2018. doi: 10.1515/tmmp-2017-0021. [Online]. Available: doi.org/10.1515/tmmp-2017-0021