小さな標数の有限体上連立二次方程式における XL アルゴリズムを用 いた解決時間の評価

田中　哲士 †‡　　鄭　振牟 ‡　　櫻井　幸一 †‡

† 九州先端科学技術研究所
814-0001 福岡県福岡市早良区百道浜２丁目１番２２号　福岡 SRP センタービル７階

‡ 九州大学
819-0395 福岡県福岡市西区元岡 744
tanasato@itslab.inf.kyushu-u.ac.jp, ccheng@imi.kyushu-u.ac.jp,
sakurai@csce.kyushu-u.ac.jp

あらまし 有限体上の連立二次方程式の求解問題（MQ 問題）は多変数公開鍵暗号の安全性の根拠となっている．XL（eXtended Linearization）アルゴリズムは MQ 問題を解決するためのアルゴリズムのひとつであり，MQ 問題の解決困難性を評価する重要な指標として考えられている．本研究では，GPU 上を用いた XL-Wiedemann アルゴリズムの実装を示す．我々の実装では GF(2) 上の 37 変数 74 方程式の MQ 問題を 36,972 秒，GF(3) 上の 24 変数 48 方程式の MQ 問題を 933 秒，GF(5) 上の 21 変数 42 方程式の MQ を 347 秒で解決可能である．

Evaluating Solving Time of Multivariate Quadratic Equation System using XL Algorithm over Small Finite Fields

Satoshi Tanaka†‡　　Chen-Mou Cheng‡　　Kouichi Sakurai†‡

†Institute of Systems, Information Technologies and Nanotechnologies.
Fukuoka SRP Center Building 7F, 2-1-22, Momochihama, Sawara-ku, Fukuoka, 814-0001, JAPAN

‡Kyushu University
744 Motooka, Nishi-ku, Fukuoka 819-0395, JAPAN
tanasato@itslab.inf.kyushu-u.ac.jp, ccheng@imi.kyushu-u.ac.jp,
sakurai@csce.kyushu-u.ac.jp

Abstract The security of multivariate public-key system is based on the problem of solving multivariate quadratic equation systems over finite fields (MQ problem). The XL (eXtended Linearization) is a aolving algorithm of MQ problem, and its running time is an important index of the complexity of MQ problem. In this work, we provide parallelized XL-Wiedeman algorithm on Graphics Processing Units (GPU). Our implementations solve MQ of 37 unknowns and 74 equations over GF(2) in 36,972 seconds, of 24 unknowns and 48 equations over GF(3) in 933 seconds, and of 21 unknowns and 42 equations over GF(5) in 347 seconds.

1 Introduction

The problem of finding roots of non-linear multivariate polynomial systems over finite fields is a core of the security for Multivariate public-key cryptography (MPKC). Some MPKCs (e.g. Unbalanced Oil and Vinegar scheme [8], Hidden Field Equations [10], QUAD stream cipher [5]) use the quadratic case of such problems (called MQ). Therefore, evaluating the complexity of MQ is important for these MPKCs.

There are two known algorithms for solving the MQ problem. One is the Gröbner basis method and the other is the eXtended Linearization (XL) algorithm. Both algorithms generate new equations from original systems. Although, XL is proved that is a redundant variant of a Gröbner basis algorithm F_4[3], it has advantages of memory size in practics[12].

The heaviest part of XL is the solving step of linearized systems. The Wiedemann algorithm solves $N \times N$ non-singular matrix systems, which row sparsity is k, in $O(kN^2)$ multiplications and additions. N is decided by the degree of regularity for the MQ.

1.1 Related works

There are several implementations of the XL-Wiedemann algorithm. Yang et al. evaluate the solving time of MQ instances (6-15 unknowns) by the C++ version[12]. Moreover, they show that the expected time of the MQ instance of 20 unknowns in 40 equations over GF(2^8) is in 2^{45} cycles. Cheng et al. implement on a NUMA machine and a cluster of PCs[6]. As a result, they solve MQ of 36 unknowns and 36 equations over GF(2) in 46,944 seconds, of 32 unknowns and 64 equations over GF(2^4) in 244,338 seconds and of 29 unknowns and 58 equations over GF(31) in 12,713 seconds.

1.2 Challenging issue

The Graphics Processing Units (GPU) implementations: some steps of the XL-Wiedemann algorithm can be parallelized. Therefore, we can consider that accelerating by GPU implementations. However, GPUs have different limitations from CPU implementations. Hence, we should consider how implement the XL-Wiedemann algorithm.

1.3 Our contibution

We provide GPU implementations of the XL-Wiedemann algorithm. We parallelized products of a sparse matrix and a dense vector on GPU. Moreover, we provide using the cuSPARSE library version (with floating point values) of the XL-Wiedemann algorithm. Finally, we show the experimental result of solving MQ instances over GF(2), GF(3) and GF(5). Our implementations solve MQ of 37 unknowns and 74 equations over GF(2) in 36,972 seconds, of 24 unknowns and 48 equations over GF(3) in 933 seconds, and of 21 unknowns and 42 equations over GF(5) in 347 seconds.

2 The MQ problem

The security of MPKC is based on the complexity of solving a system of multivariate non-linear equations over finite fields. The MQ problem is a quadratic case of this problem. MQ is known to be NR-complete [4].

Let $q = p^k$, where p is a prime, and $\boldsymbol{x} = \{x_1, \ldots, x_n\}$ ($\forall i$, $x_i \in \mathrm{GF}(q)$). Generally, multivariate quadratic polynomial equations of n unknowns over GF)(q) are described by the following:

$$f(\boldsymbol{x}) = \sum_{1 \le i \le j \le n} \alpha_{i,j} x_i x_j + \sum_{1 \le i \le n} \beta_i x_i + \gamma = 0, \tag{1}$$

where $\forall i, j$, $\alpha_{i,j}, \beta_i, \gamma \in \mathrm{GF}(q)$. The MQ problem consists solving quadratic polynomial equations given by $\boldsymbol{y} = \{f_1(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})\}$.

3 The XL algorithm

The original XL algorithm was proposed by Courtois in 2000[7]. The idea of XL is based on the linearization technique. Linearization is subtitution new unknowns non-linear terms (e.g. $x_1 x_2 = y_{1,2}$). If the number of equations is greater than the number of variables, it solves the system by algebraic methods (e.g. the Gaussian elimination). If not, it generates new equations from the original ones. The XL algorithm is described in Algorithm 1. The degree of regularity D is the minimal degree, which the number of linearly independent equations exceeds the number of unknowns in linearized system.

Algorithm 1 The XL algorithm[7]

Input: m quadratic polynomial equations $F = \{f_1, \ldots, f_m\}$, m-th vector $\boldsymbol{y} = F(\boldsymbol{x})$, and the degree of regularity D.

Output: the n-th unknown vector $\boldsymbol{x} = \{x_1, \ldots, x_n\}$.

1: Multiply: Generate all the product of polynomial equations and products of unknowns $\prod_{j=1}^{D-2} x_{i_j}$.
2: Linearize: Consider each monomial in the x_i of degree $\leq D$ as a new unknown and perform an elimination algorithm on the equations obtained in 1 and derive univariate equations.
3: Solve: Solve univariate equations obtained in 2 over $\mathrm{GF}(q)$.
4: Repeat: Simplify the equations and repeat the process to find the values of other unknowns.

The XL algorithm generates sparse equations in the multiplication step. The number of non-zero terms of an equation is only $\binom{n+2}{2}$ (of $\binom{n+D}{D}$ terms), since generated equations are just producted of original equations and monomials. However, the Gaussian elimination is not suited to solve systems of sparse linear equations. it is quantitative for the size of a matrix. The XL-Wiedemann algorithm[9] improved this disadvantage of the original XL by replacing the Gaussian elimination with the Wiedemann algorithm[11], which is suited to a system of sparse linear equations.

3.1 The Wiedemann algorithm

The Wiedemann algorithm[11] is a solving method for a system of linear sparse equations over finite fields. Let A is an $N \times N$ non-singular matrix over $\mathrm{GF}(q)$. The Wiedemann algorithm finds a n-th non-zero vector $\mathbf{x}$, where $\mathbf{y} = \mathbf{A}\mathbf{x}$. The block Wiedemann algorithm is described in Algorithm 2.

Algorithm 2 The Wiedemann algorithm[11]

Input: $N \times N$ non-singular matrix A and the N-th vector $\boldsymbol{b}$, where $A\boldsymbol{x} = \boldsymbol{b}$.

Output: the N-th unknown vector $\boldsymbol{x}$.

1: Set $\boldsymbol{b}_0 = \boldsymbol{b}$, $k = 0$. $\boldsymbol{y}_0 = 0$ and $d_0 = 0$.
2: Compute the matrix sequence $s_i = \boldsymbol{u}_{k+1} A^i \boldsymbol{b}_k$ for $0 \leq i \leq 2(N - d)$, with a random N-th vector u_{k+1}.
3: Set $f(\lambda)$ to the minimum polynomial of the sequence of s_i with the Berlekamp-Massey algorithm.
4: Set $\boldsymbol{y}_{k+1} = \boldsymbol{y}_k + f^-(A)\boldsymbol{b}_k$, where $f^-(\lambda) := \frac{f(\lambda) - f(0)}{\lambda}$, $\boldsymbol{b}_{k+1} = \boldsymbol{b}_0 + A\boldsymbol{y}_{k+1}$ and $d_{k+1} = d_k + \deg f(\lambda)$.
5: If $\boldsymbol{b}_{k+1} = 0$, then the solution is $\boldsymbol{x} = \boldsymbol{y}_k$
6: Set $k = k + 1$ and go to step 2.

3.2 Sparse matrix forms

We assume that D is the degree of regularity for the XL algorithm. Then, XL constructs $\binom{n+D}{D} \times \binom{n+D}{D}$ linearized matrix from

MQ instances of n unknowns and m equations over GF(q). However, quadratic polynomial equations of n unkowns have only $\binom{n+2}{2}$ terms. Therefore, we can reduce computations of matrix-vector product and the memory size of matrix by using sparse matrix form.

Let N be the degree of row and column in a matrix(i.e. $N \times N$ matrix), and num_{NZ} be the number of non-zero elements in the matrix. Sparse matrix forms have value, row-index and column-index data of non-zero elements in a matrix. There are some sparse matrix foratss as the following[1]:
The COO (coordinate) format is the most basic. It holds simplly value, row-index and column-index data of non-zero elements in the matrix. Therefore, it requires $3num_{NZ}$ for the emory space.
The CSR (compressed storage row) assumes that the data vector is ordered by the row-index. It differs only row-index from the COO formats, it holds the head number of non-zero terms in each row-vector of the matrix instead of row-index data. The, it requires $2num_{NZ} + N$.
The ELL (Ellpack-Itpack) format uses two dense $N \times max_{NZ}$ matrices, where max_{NZ} is the maximal number of non-zero terms in a row-vector. One matrix shows the value of non-zero matrix, and the other shows the column-index. Figure 1 shows the example of each foramt.

4 CUDA API

CUDA is a development environment for GPU, based on C language and provided by NVIDIA. Proprietary tools for using GPU have existed before CUDA was proposed. However, such tools as OpenGL and DirectX need to output computer graphics while processing work. Therefore, these tools are not efficient. CUDA is efficient, because CUDA uses computational core of GPU directly.

In CUDA, hosts correspond to computers, and devices correspond to graphic cards. CUDA works by making the host control the device. Kernel is a function the host used to control the device. Because only one kernel can work at a time, a program requires parallelizing processes in a kernel. A kernel handles some blocks in parallel. A block also handles some threads in parallel. Therefore a kernel can handle many threads simultaneously.

4.1 cuSPARSE library

NVIDIA provides several libraries for linear algebra. For example, the cuBLAS library provides functions of the Basic Linear Algebra Subprograms (BLAS) library. BLAS classfies three levels of functions. Level 1 functions gives operations of vectors and vectors, level 2 acheives operations vectors and matrices, and level 3 allows matrix and matrix operations. Actually, the cuSPARSE library is the sparse matrix version of the cuBLAS library. Therefore, cuSPARSE also provides three level functons.

5 The XL-Wiedemann algorithm on GPU

5.1 Degree of regurarity over small fields

The heviest point of the XL-Wiedeman algorithm is solving $N \times N$ matrix systems as a linear algebra. In XL, N is decided by the degree of regularity D as $N = \binom{N+D}{D}$. The dgeree of regularity is the minimal degree, where the number of linearly independent equations exceeds the number of linearized unknowns. We can figure the number of linearized unknowns N for the degree d as $N = \binom{N+d}{d}$ easily. Rønjon and Raddum gives that the upperbound for

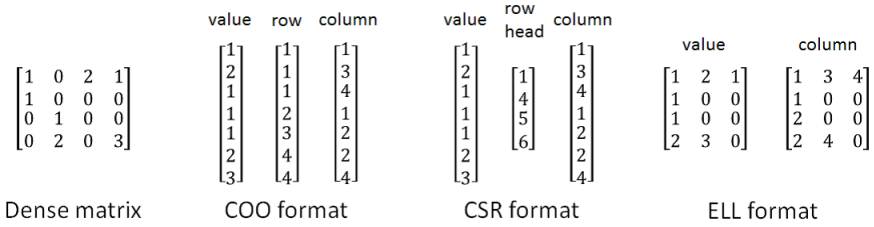

Figure 1: Image of each sparse matrix formats.

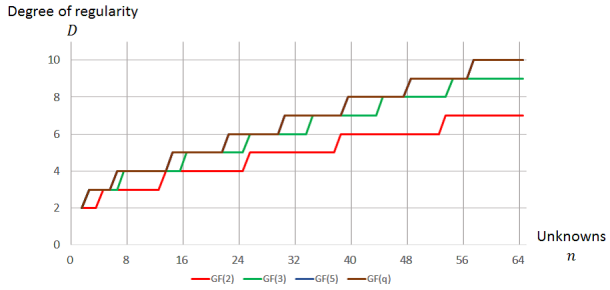

Figure 2: The deree of regularity for $m = 2n$ cases, under $n \leq 64$.

the number of linearly independent equations I is decided by the following formula:

$$I = \sum_{i=0}^{\frac{D_m}{D_e}} (-1)^i \binom{m+i}{i+1} \sum_{j=0}^{D_m - i \cdot D_e} \binom{n}{j}. \quad (2)$$

Then, D_m is the maximal degree of monomials multplying equations, and D_e is the degree of the original equations system. For the MQ problem, $D_m = D - 2$ and $D_e = 2$. Therefore, we can find the minimal degree D, where $I \geq N (= \binom{N+D}{D})$ by Formula (2). Figure 2 shows degrees of regularity for MQ of n unknowns and $2n$ equations over GF(2), GF(3), GF(5) and other prime fields under $n \leq 64$. Acturally, the cases of GF(5) and other prime fields are similar. $textGF(2)$ and $textGF(3)$ differ from other fields, because we consider reductions by field equation $\alpha^q = \alpha$ ($\alpha \in \mathrm{GF}(q)$).

5.2 Choosing equations

By the definition of the degree of regularity, $I \geq N$. Then, we get an $I \times N$ matrix by the extend step of the XL algorithm. For the Wiedemann algorthm, we should reduce to N from I. The simplest way is removing equations by random choosing.

5.3 The Wiedemann algorithm

Mainly, the Wiedemann algorithm is seperated to three steps. The first step is generating the sequence $\{(\boldsymbol{u}, A^i\boldsymbol{b})\}_{i=0}^{2N}$ for a $N \times N$ matrix A, a vector $\boldsymbol{b}$, where $A\boldsymbol{x} = \boldsymbol{b}$ and random vector $\boldsymbol{u}$. The second step is finding the minimal polynomial of the generated sequence $f(\lambda)$ by the Berlekamp-Massey algorithm. The final step is compute $f^-(A)\boldsymbol{b}$, where $f^-(\lambda) = \frac{f(\lambda)-f(0)}{\lambda}$. In this work, we only implement the first step and the final step on GPU. Because, the Berlekamp-Massey algorithm is very sequencial (it seems no parallelizable) and has many conditional branchs. Since, both of the sequencial algorithm and the conditional branch are not suitable for GPU, we implement the second step on CPU.

5.4 Generating sequence $\{(\boldsymbol{u}, A^i\boldsymbol{b})\}_{i=0}^{2N}$

This step requires products the sparse matrix A and the dense vector $A^{i-1}\boldsymbol{b}$, and dot products $(\boldsymbol{u}, A^i\boldsymbol{b})$. However, we can choose the random vector $\boldsymbol{u}$ as $\boldsymbol{u} = \{1, 0, \ldots, 0\}$. Therefore, dot products can be computed by looking up the first element of the vector $A^i\boldsymbol{b}$. Hence, we should consider only producs of the sparse matrix A and the dense vector $A^{i-1}\boldsymbol{b}$

Products of the sparse matrix A and the dense vector $A^{i-1}\boldsymbol{b}$ has two steps. The first one is multiplications of non-zero elements in the matrix and elements in the vector. The other is summations of multiplication result for each row.

We choose the ELL format for sparse matrices. One of advantages of this format is every column width is same in a matrix and multiplication result holds such width. In CUDA kernels (GPU functions), the column width can correspond with the number of threads of the kernel and the row height corresponds the number of blocks. Usually, each blocks has the same number of threads. Therefore, the ELL format is suited to CUDA kernels.

In summations of multiplication result, we use the parallel reduction technique [2]. This technique computes summations of n terms in $\log n$ steps.

5.5 Computing $f^{-}(A)\boldsymbol{b}$

Since $f^{-}(A)\boldsymbol{b} = \sum_{i=1}^{d} c_i A^{i-1}\boldsymbol{b}$, where d is the degree of $f(\lambda)$, this step is summations of $c_i A^{i-1}\boldsymbol{b}$. Then, $A^i\boldsymbol{b}$ is similar to the first step of the Wiedemann algorithm. Hence, there is two strategies for $A^i\boldsymbol{b}$. One is storing the result of $A^i\boldsymbol{b}$ on GPU. This strategy can reduce recomputations of $A^i\boldsymbol{b}$. However, it needs about N^2 memory spaces for $A^i\boldsymbol{b}$, where $0 \leq i \leq N$ (since $d \leq N$). Therefore, this strategy can be used only small matrix cases.

The other is recomputing $A^i\boldsymbol{b}$. Although, it requires more d products of $A^i\boldsymbol{b}$, it needs memory space only $A^{i-1}\boldsymbol{b}$ (last vector of $A^i\boldsymbol{b}$). Terefore, this strategy is suitable for large matrix cases.

5.6 cuSPARSE version

The cuSPARSE library provides functions of products a sparse matrix and a dense vector. Therefore, using cuSPARSE is another choice for products of A and $A^{i-1}\boldsymbol{b}$. There are two important points for implementations. One is the function form. The cuSPARSE library only provides $\boldsymbol{y} \leftarrow \alpha A\boldsymbol{x} + \beta\boldsymbol{y}$ (A: matrix, $\boldsymbol{x}$, $\boldsymbol{y}$: vector and α, β: scalar) form functions for the CSR format. Then, for the first step, we set $\beta = 0$. Moreover, in the cuSPARSE version, we should use the CSR format for sparse matrices.

The other is the type of variables. The cuSPARSE library suportes only floating point values (does not support integer values). It means that the cuSPARSE library does not directly supporte any field operations. Then, we should coordinate cuSPARSE functions as field operations by additional operations (e.g. modular operations).

6 Experimentation

We implement the XL-Wiedemann algorithm on GPU. Our implementations are two types, integer version and the cuSPARSE (floating point) version. We solve the largest case of $D = 4, 5$, over GF(2), GF(3) and GF(5) by both XL-Wiedemann implementations. Table 1 shows the detail of each MQ construction.

Table 2 shows the experimental result, and Table 3 shows the profile of the Wiedemann algorithm. The cuSPARSE library seems to be better choice for larger case. In our experimentations, the Berlekamp-Massey algorithm is heavy for the XL-Wiedemann algorithm. However, it is not problem, because we can choose faster libraries on CPU like MAGMA.

7 Conclusion

We provide GPU implementations of the XL-Wiedemann algorithm. Also, we show the two

Table 1: Constructions of MQ instances.

Field GF(q)	GF(2)		GF(3)		GF(5)	
Degree of regularity D	4	5	4	5	4	5
Unknowns n	24	37	15	24	13	21
Equations m	48	74	30	48	26	42
Matrix						
Linearized terms	12,950	510,415	3,635	110,954	2,379	65,758
Nonzero terms	301	704	136	325	105	253

Table 2: Result of XL-Wiedemann on GPU.

	Field GF(q)	GF(2)		GF(3)		GF(5)	
	Degree of regularity D	4	5	4	5	4	5
	Unknowns n	24	37	15	24	13	21
	Equations m	48	74	30	48	26	42
Integer	Solving time (sec)	14.7358	83,782.11	0.5847	2,089.30	0.4415	601.124
	Extension (sec)	0.1248	130.98	0.0116	7.29	0.0059	3.347
	Wiedemann (sec)	14.6101	83,651.08	0.5729	2,082.01	0.4355	597.777
cuSPARSE	Solving time (sec)	8.8982	36,971.85	0.8684	932.95	0.4852	346.571
	Extension (sec)	0.0885	128.28	0.0098	8.00	0.0050	3.366
	Wiedemann (sec)	8.8077	36,843.49	0.8583	924.95	0.4800	343.204

Table 3: Profile of the Wiedemann slgorithm.

	Field GF(q)	GF(2)		GF(3)		GF(5)	
	Degree of regularity D	4	5	4	5	4	5
	Unknowns n	24	37	15	24	13	21
	Equations m	48	74	30	48	26	42
Integer	Running time (sec)						
	Wiedemann	14.6101	83,651.08	0.5729	2,082.01	0.4355	597.777
	Generating Sequence	9.5806	49,719.75	0.3030	1,104.82	0.2131	302.236
	Berlekamp-Massey	4.9253	9,035.16	0.2379	439.1057	0.19	148.328
	Computing $f^-(A)\boldsymbol{b}$	0.0937	24,895.43	0.0305	537.99	0.0273	147.188
	Memory Usage (MB)						
	Matrix	29.74	2741.49	5.66	412.67	2.86	190.39
	Stream	1279.47	0	100.81	0	43.22	0
cuSPARSE	Running time (sec)						
	Wiedemann	8.8077	36,843.49	0.8583	924.94	0.4800	343.2035
	Generating Sequence	3.8079	22,215.69	0.4284	325.75	0.2418	108.0073
	Berlekamp-Massey	4.8855	9,059.83	0.4284	325.75	0.1999	183.685
	Computing $f^-(A)\boldsymbol{b}$	0.1045	5,567.20	0.0403	160.77	0.0372	51.473
	Memory Usage (MB)						
	Matrix	44.66	4114.18	5.67	413.10	2.87	190.64
	Stream	1279.47	0	100.81	0	43.22	0

types, integer case and using cuSPARSE library (floating point values) case. Our implementations solve MQ of 37 unknowns and 74 equations over GF(2) in 36,972 seconds, of 24 unknowns and 48 equations over GF(3) in 933 seconds, and of 21 unknowns and 42 equations over GF(5) in 347 seconds by using cuSPARSE library case. Our further goal is evaluating the expected time of larger degree cases (e.g. the case of $D = 6$).

Acknowledgement

This work is partly supported by "Study on Secure Cryptosystem using Multivariate polynomial," no. 0159-0172, Strategic Information and Communications R&D Promotion Programme (SCOPE), the Ministry of Internal Affairs and Communications, Japan and Grant-in-Aid for Young Scientists (B), Grant number 24740078.

References

[1] cusparse::cuda toolkit documentation. `http://docs.nvidia.com/cuda/cusparse`, Accessed August 2014.

[2] Optimizing parallel reduction in cuda. `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf`, Accessed August 2014.

[3] Gwénolé Ars, Jean-Charles Faugere, Hideki Imai, Mitsuru Kawazoe, and Makoto Sugita. Comparison between xl and gröbner basis algorithms. In *Advances in Cryptology-ASIACRYPT 2004*, pages 338–353. Springer, 2004.

[4] Gregory V. Bard. *Algebraic Cryptanalysis*. Springer, 2009.

[5] Côme Berbain, Henri Gilbert, and Jacques Patarin. Quad: A practical stream cipher with provable security. In *Advances in Cryptology-EUROCRYPT 2006*, pages 109–128. Springer, 2006.

[6] Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. Solving quadratic equations with xl on parallel architectures. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 356–373. Springer, 2012.

[7] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial equations. In *Advances in Cryptology-EUROCRYPT 2000*, pages 392–407. Springer, 2000.

[8] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In *Advances in Cryptology - EUROCRYPT '99*, pages 206–222. Springer, 1999.

[9] Wael Said Abdelmageed Mohamed, Jintai Ding, Thorsten Kleinjung, Stanislav Bulygin, and Johannes Buchmann. Pwxl: A parallel wiedemann-xl algorithm for solving polynomial equations over $gf(2)$. *SCC*, 2010:89–100, 2010.

[10] Jacques Patarin. Hidden fields equations (hfe) and isomorphisms of polynomials (ip): Two new families of asymmetric algorithms. In *Advances in Cryptology - Eurocrypt '96*, pages 33–48. Springer, 1996.

[11] Douglas Wiedemann. Solving sparse linear equations over finite fields. *Information Theory, IEEE Transactions on*, 32(1):54–62, 1986.

[12] Bo-Yin Yang, Owen Chia-Hsin Chen, Daniel J Bernstein, and Jiun-Ming Chen. Analysis of quad. In *Fast Software Encryption*, pages 290–308. Springer, 2007.