Time-Area Optimized Public-Key Engines: $\mathcal{MQ}$-Cryptosystems as Replacement for Elliptic Curves? *

Andrey Bogdanov, Thomas Eisenbarth, Andy Rupp, Christopher Wolf
Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany
{abogdanov,eisenbarth,arupp}@crypto.rub.de,
chris@Christopher-Wolf.de or cbw@hgi.rub.de

Abstract

In this paper ways to efficiently implement public-key schemes based on $\mathcal{M}$ultivariate $\mathcal{Q}$uadratic polynomials ($\mathcal{MQ}$-schemes for short) are investigated. In particular, they are claimed to resist quantum computer attacks. It is shown that such schemes can have a much better time-area product than elliptic curve cryptosystems. For instance, an optimised FPGA implementation of amended TTS is estimated to be over 50 times more efficient with respect to this parameter. Moreover, a general framework for implementing small-field $\mathcal{MQ}$-schemes in hardware is proposed which includes a systolic architecture performing Gaussian elimination over composite binary fields.

1 Introduction

Efficient implementations of public key schemes play a crucial role in numerous real-world security applications: Some of them require messages to be signed in real time (like in such safety-enhancing automotive applications as car-to-car communication), others deal with thousands of signatures per second to be generated (e.g. high-performance security servers using so-called HSMs - Hardware Security Modules). In this context, software implementations even on high-end processors can often not provide the performance level needed, hardware implementations being thus the only option. In this paper we explore the approaches to implement $\mathcal{M}$ultivariate $\mathcal{Q}$uadratic-based public-key systems in hardware meeting the requirements of efficient high-performance applications. The security of public key cryptosystems widely spread at the moment is based on the difficulty of solving a small class of problems: the RSA scheme relies on the difficulty of factoring large integers, while the hardness of computing discrete logarithms provides the basis for ElGamal, Diffie-Hellmann scheme and elliptic curves cryptography (ECC). Given that the security of all public key schemes used in practice relies on such a limited set of problems that are *currently* considered to be hard, research on new schemes based on other classes of problems is necessary as such work will provide greater diversity and hence forces cryptanalysts to spend additional effort concentrating on completely new types of problems. Moreover, we make sure that not all "crypto-eggs" are in one basket. In this context, we want to point out that important results on the potential weaknesses of existing public key schemes are emerging. In particular techniques for factorisation and solving discrete logarithms improve continually. For example, polynomial time quantum algorithms can be used to solve both problems. Therefore, the existence of quantum computers in the range of a few thousands of qbits

*This is a revised version of the original paper accepted for CHES 2008.

would be a real-world threat to systems based on factoring or the discrete logarithm problem. This emphasises the importance of research into new algorithms for asymmetric cryptography.

One proposal for secure public key schemes is based on the problem of solving $\mathcal{M}$ultivariate $\mathcal{Q}$uadratic equations ($\mathcal{MQ}$-problem) over finite fields $\mathbb{F}$, *i.e.* finding a solution vector $x \in \mathbb{F}^n$ for a given system of m polynomial equations in n variables each

$$\begin{cases} y_1 &= p_1(x_1, \ldots, x_n) \\ y_2 &= p_2(x_1, \ldots, x_n) \\ &\vdots \\ y_m &= p_m(x_1, \ldots, x_n)\,, \end{cases}$$

for given $y_1, \ldots, y_m \in \mathbb{F}$ and unknown $x_1, \ldots, x_n \in \mathbb{F}$ is difficult, namely $\mathcal{NP}$-complete. An overview over this field can be found in [14].

Roughly speaking, most work on public-key hardware architectures tries to optimise either the speed of a single instance of an algorithm (e.g., high-speed ECC or RSA implementations) or to build the smallest possible realization of a scheme (e.g., lightweight ECC engine). A major goal in high-performance applications is, however, in addition to pure time efficiency, an optimised cost-performance ratio. In the case of hardware implementations, which are often the only solution in such scenarios, costs (measured in chip area and power consumption) is roughly proportional to the number of logic elements (gates, FPGA slices) needed. A major finding of this paper is that $\mathcal{MQ}$-schemes have the better time-area product than established public key schemes. This holds, interestingly, also if compared to elliptic curve schemes, which have the reputation of being particularly efficient.

The first public hardware implementation of a cryptosystem based on multivariate polynomials we are aware of is [17], where enTTS is realized. A more recent result on the evaluation of hardware performance for Rainbow can be found in [2].

1.1 Our Contribution

Our contribution is many-fold. First, a clear taxonomy of secure multivariate systems and existing attacks is given. Second, we present a systolic architecture implementing Gauss-Jordan elimination over GF(2^k) which is based on the work in [13]. The performance of this central operation is important for the overall efficiency of multivariate based signature systems. Then, a number of concrete hardware architectures are presented having a low time-area product. Here we address both rather conservative schemes such as UOV as well as more aggressively designed proposals such as Rainbow or amended TTS (amTTS). For instance, an optimised implementation of amTTS is estimated to have a TA-product over 50 times lower than some of the most efficient ECC implementations. Moreover, we suggest a generic hardware architecture capable of computing signatures for the wide class of multivariate polynomial systems based on small finite fields. This generic hardware design allows us to achieve a time-area product for UOV which is somewhat smaller than that for ECC, being considerably smaller for the short-message variant of UOV.

2 Foundations of $\mathcal{MQ}$-Systems

In this section, we introduce some properties and notations useful for the remainder of this article. After briefly introducing $\mathcal{MQ}$-systems, we explain our choice of signature schemes and give a brief description of them.

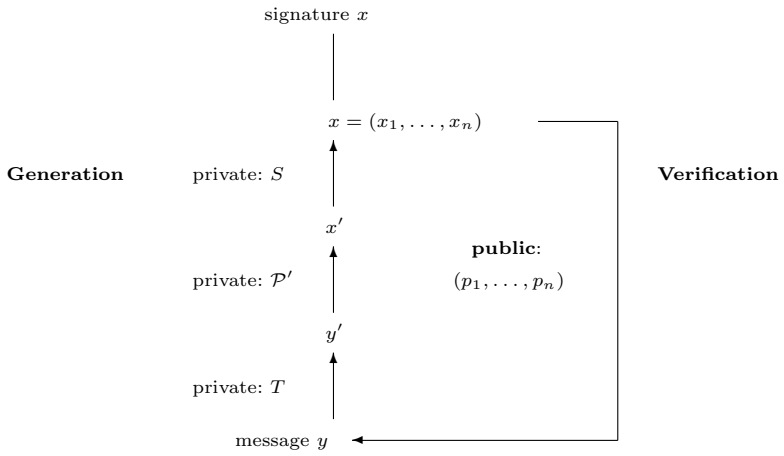

Figure 1: Graphical Representation of the $\mathcal{MQ}$-trapdoor $(S, \mathcal{P}', T)$

2.1 Mathematical Background

Let $\mathbb{F}$ be a finite field with $q := |\mathbb{F}|$ elements and define $\mathcal{M}$ultivariate $\mathcal{Q}$uadratic ($\mathcal{MQ}$) polynomials p_i of the form

$$p_i(x_1, \ldots, x_n) := \sum_{1 \le j \le k \le n} \gamma_{i,j,k} x_j x_k + \sum_{j=1}^{n} \beta_{i,j} x_j + \alpha_i \,,$$

for $1 \le i \le m; 1 \le j \le k \le n$ and $\alpha_i, \beta_{i,j}, \gamma_{i,j,k} \in \mathbb{F}$ (constant, linear, and quadratic terms). We now define the polynomial-vector $\mathcal{P} := (p_1, \ldots, p_m)$ which yields the public key of these $\mathcal{M}$ultivariate $\mathcal{Q}$uadratic systems. This public vector is used for signature verification. Moreover, the private key (cf Fig.1) consists of the triple $(S, \mathcal{P}', T)$ where $S \in \mathrm{Aff}(\mathbb{F}^n), T \in \mathrm{Aff}(\mathbb{F}^m)$ are affine transformations and $\mathcal{P}' \in \mathcal{MQ}(\mathbb{F}^n, \mathbb{F}^m)$ is a polynomial-vector $\mathcal{P}' := (p'_1, \ldots, p'_m)$ with m components; each component is in $x'_1, \ldots, x'_n$. Throughout this paper, we will denote components of this private vector $\mathcal{P}'$ by a prime $'$. The linear transformations S and T can be represented in the form of invertible matrices $M_S \in \mathbb{F}^{n \times n}, M_T \in \mathbb{F}^{m \times m}$, and vectors $v_S \in \mathbb{F}^n, v_T \in \mathbb{F}^m$ *i.e.* we have $S(x) := M_S x + v_S$ and $T(x) := M_T x + v_T$, respectively. In contrast to the public polynomial vector $\mathcal{P} \in \mathcal{MQ}(\mathbb{F}^n, \mathbb{F}^m)$, our design goal is that the private polynomial vector $\mathcal{P}'$ does allow an efficient computation of $x'_1, \ldots, x'_n$ for given $y'_1, \ldots, y'_m$. At least for secure $\mathcal{MQ}$-schemes, this is not the case if the public key $\mathcal{P}$ alone is given. The main difference between $\mathcal{MQ}$-schemes lies in their special construction of the central equations $\mathcal{P}'$ and consequently the trapdoor they embed into a specific class of $\mathcal{MQ}$-problems.

In this kind of schemes, the public key $\mathcal{P}$ is computed as function composition of the affine transformations $S : \mathbb{F}^n \to \mathbb{F}^n$, $T : \mathbb{F}^m \to \mathbb{F}^m$ and the central equations $\mathcal{P}' : \mathbb{F}^n \to \mathbb{F}^m$, *i.e.* we have $\mathcal{P} = T \circ \mathcal{P}' \circ S$. To fix notation further, we note that we have $\mathcal{P}, \mathcal{P}' \in \mathcal{MQ}(\mathbb{F}^n, \mathbb{F}^m)$, *i.e.* both are functions from the vector space $\mathbb{F}^n$ to the vector space $\mathbb{F}^m$. By construction, we have $\forall x \in \mathbb{F}^n : \mathcal{P}(x) = T(\mathcal{P}'(S(x)))$.

2.2 Signing

To sign for a given $y \in \mathbb{F}^m$, we observe that we have to invert the computation of $y = \mathcal{P}(x)$. Using the trapdoor-information $(S, \mathcal{P}', T)$, cf Fig. 1, this is easy. First, we observe that transformation T is a bijection. In particular, we can compute $y' = M_T^{-1} y$. The same is true for given $x' \in \mathbb{F}^n$ and $S \in \mathrm{Aff}(\mathbb{F}^n)$. Using the LU-decomposition of the matrices M_S, M_T, this computation takes

time $O(n^2)$ and $O(m^2)$, respectively. Hence, the difficulty lies in evaluating $x' = \mathcal{P}'^{-1}(y')$. We will discuss strategies for different central systems $\mathcal{P}'$ in Sect. 2.4.

2.3 Verification

In contrast to signing, the verification step is the same for all $\mathcal{MQ}$-schemes and also rather cheap, computationally speaking: given a pair $x \in \mathbb{F}^n, y \in \mathbb{F}^m$, we evaluate the polynomials

$$p_i(x_1, \ldots, x_n) := \sum_{1 \le j \le k \le n} \gamma_{i,j,k} x_j x_k + \sum_{j=1}^{n} \beta_{i,j} x_j + \alpha_i \,,$$

for $1 \le i \le m; 1 \le j \le k \le n$ and given $\alpha_i, \beta_{i,j}, \gamma_{i,j,k} \in \mathbb{F}$. Then, we verify that $p_i = y_i$ holds for all $i \in \{1, \ldots, m\}$. Obviously, all operations can be efficiently computed. The total number of operations takes time $O(mn^2)$.

2.4 Description of the Selected Systems

Based on [14] and some newer results, we have selected the following suitable candidates for efficient implementation of signature schemes: enhanced TTS, amended TTS, Unbalanced Oil and Vinegar and Rainbow. Systems of the big-field classes HFE (Hidden Field Equations), MIA (Matsumoto Imai Scheme A) and the mixed-field class ℓIC — ℓ-Invertible Cycle [8] were excluded as results from their software implementation show that they cannot be implemented as efficiently as schemes from the small-field classes, *i.e.* enTTS, amTTS, UOV and Rainbow. The proposed schemes and parameters are summarised in Table 1.

Table 1: Proposed Schemes and Parameters

	q	n	m	τ	K	Solver
Unbalanced Oil	256	30	10	0.003922	10	$1 \times K = 10$
and Vinegar (UOV)		60	20		20	$1 \times K = 20$
Rainbow	256	42	24	0.007828	12	$2 \times K = 12$
enhanced TTS (v1)	256	28	20	0.000153	9	$2 \times K = 9$
(v2)				0.007828	10	$2 \times K = 10$
amended TTS	256	34	24	0.011718	4,10	$1 \times K = 4$, $2 \times K = 10$

2.4.1 Unbalanced Oil and Vinegar (UOV).

$$p'_i(x'_1, \ldots, x'_n) := \sum_{j=1}^{n-m} \sum_{k=j}^{n} \gamma'_{i,j,k} x'_j x'_k \text{ for } i = 1 \ldots v_1$$

Unbalanced Oil and Vinegar Schemes were introduced in [10, 11]. Here we have $\gamma \in \mathbb{F}$, *i.e.* the polynomials p are over the finite field $\mathbb{F}$. In this context, the variables x'_i for $1 \le i \le n-m$ are called the "vinegar" variables and x'_i for $n - m < i \le n$ the "oil" variables. We also write $o := m$ for the number of oil variables and $v := n - m = n - o$ for the number of vinegar variables. To invert UOV, we need to assign random values to the vinegar variables $x'_1, \ldots, x'_v$ and obtain a linear system in the oil variables $x'_{v+1}, \ldots, x'_n$. All in all, we need to solve a $m \times m$ system and have hence $K = m$. The probability that we do *not* obtain a solution for this system is $\tau^{UOV} = 1 - \frac{\prod_{i=0}^{m-1} q^m - q^i}{q^{m^2}}$ as there are q^{m^2} matrices over the finite field $\mathbb{F}$ with $q := |\mathbb{F}|$ elements and $\prod_{i=0}^{m-1} q^m - q^i$ invertible ones [14].

Taking the currently known attacks into account, we derive the following secure choice of parameters for a security level of 2^{80}:

- Small datagrams: $m = 10, n = 30, \tau \approx 0.003922$ and one $K = 10$ solver

- Hash values: $m = 20, n = 60, \tau \approx 0.003922$ and one $K = 20$ solver

The security has been evaluated using the formula $O(q^{v-m-1}m^4) = O(q^{n-2m-1}m^4)$. Note that the first version (*i.e.* $m = 10$) can only be used with messages of less than 80 bits. However, such datagrams occur frequently in applications with power or bandwidth restrictions, hence we have noted this special possibility here.

2.4.2 Rainbow.

Rainbow is the name for a generalisation of UOV [7]. In particular, we do not have one layer, but several layers. This way, we can reduce the number of variables and hence obtain a faster scheme when dealing with hash values. The general form of the Rainbow central map is given below.

$$p'_i(x'_1, \ldots, x'_n) \quad := \quad \sum_{j=1}^{v_l} \sum_{k=j}^{v_{l+1}} \gamma'_{i,j,k} x'_j x'_k \text{ for } i = v_l \ldots v_{l+1}, 1 \leq l \leq L$$

We have the coefficients $\gamma \in \mathbb{F}$, the layers $L \in \mathbb{N}$ and the vinegar splits $v_1 < \ldots < v_{L+1} \in \mathbb{N}$ with $n = v_{L+1}$. To invert Rainbow, we follow the strategy for UOV — but now layer for layer, *i.e.* we pick random values for $x_1, \ldots, x_{v_1}$, solve the first layer with an $(v_2 - v_1) \times (v_2 - v_1)$-solver for $x_{v_1+1}, \ldots, x_{v_2}$, insert the values $x_1, \ldots, x_{v_2}$ into the second layer, solve second layer with an $(v_3 - v_2) \times (v_3 - v_2)$-solver for $x_{v_2+1}, \ldots, x_{v_3}$ until the last layer L. All in all, we need to solve sequentially L times $(v_l - v_{l-1}) \times (v_l - v_{l-1})$ systems for $l = 2 \ldots L + 1$. The probability that we do not obtain a solution for this system is $\tau^{rainbow} = 1 - \prod_{l=1}^{L} \frac{\prod_{i=0}^{v_{l+1}-v_l} q^{v_{l+1}-v_l} - q^i}{q^{v_{l+1}-v_l^2}}$ using a similar argument as in Sec. 2.4.1.

Taking the latest attack from [3] into account, we obtain the parameters $L = 2, v_1 = 18, v_2 = 30, v_3 = 42$ for a security level of 2^{80}, *i.e.* a two layer scheme 18 initial vinegar variables and 12 equations in the first layer and 12 new vinegar variables and 12 equations in the second layer. Hence, we need two $K = 12$ solvers and obtain $\tau \approx 0.007828$

2.4.3 amended TTS (amTTS).

The central polynomials $\mathcal{P}' \in \mathcal{MQ}(\mathbb{F}^n, \mathbb{F}^m)$ for $m = 24, n = 34$ in amTTS [6] are defined as given below:

$$\begin{aligned}
p'_i &:= x'_i + \alpha'_i x'_{\sigma(i)} + \sum_{j=1}^{8} \gamma'_{i,j} x'_{j+1} x'_{11+(i+j \bmod 10)} \text{ , for } i = 10 \ldots 19; \\
p'_i &:= x'_i + \alpha'_i x'_{\sigma(i)} + \gamma'_{0,i} x'_1 x'_i + \sum_{j=1}^{8} \gamma'_{i,j} x'_{15+(i+j+4 \bmod 8)j+1} x'_{\pi(i,j)} \text{ , for } i = 20 \ldots 23; \\
p'_i &:= x'_i + \gamma'_{0,i} x'_0 x'_i + \sum_{j=1}^{9} \gamma'_{i,j} x'_{24+(i+j+6 \bmod 10)j+1} x'_{\pi(i,j)} \text{ , for } i = 24 \ldots 33.
\end{aligned}$$

We have $\alpha, \gamma \in \mathbb{F}$ and σ, π permutations, *i.e.* all polynomials are over the finite field $\mathbb{F}$. We see that they are similar to the equations of Rainbow (Sec. 2.4.2) — but this time with sparse polynomials. Unfortunately, there are no more conditions given on σ, π in [6] — we have hence picked one suitable permutation for our implementation.

To invert amTTS, we follow the sames ideas as for Rainbow — except with the difference that we have to invert twice a 10×10 system ($i = 10 \ldots 19$ and $24 \ldots 33$) and once a 4×4 system, *i.e.* we have $K = 10$ and $K = 4$. Due to the structure of the equations, the probability for *not* getting a solution here is the same as for a 3-Layer Rainbow scheme with $v_1 = 10, v_2 = 20, v_3 = 24, v_4 = 34$ variables, *i.e.* $\tau^{amTTS} = \tau^{Rainbow}(10, 20, 24, 34) \approx 0.011718$.

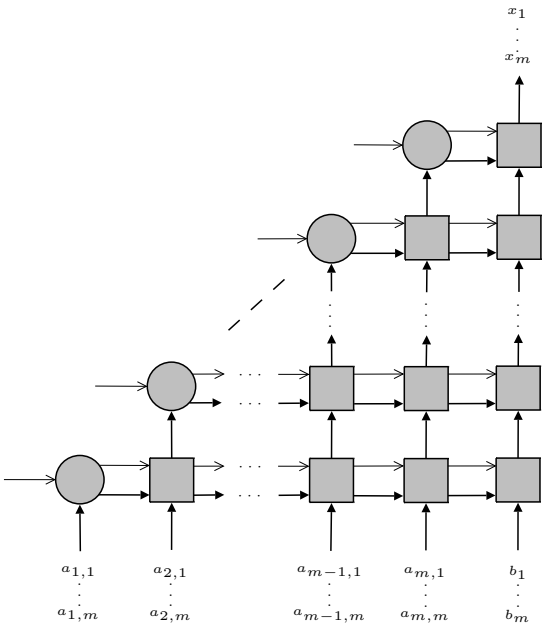

Figure 2: Signature Core Building Block: Systolic Array LSE Solver (Structure)

2.4.4 enhanced TTS (enTTS).

The overall idea of enTTS is similar to amTTS, $m = 20, n = 28$. For a detailed description of enTTS see [16, 15]. According to [6], enhanced TTS is broken, hence we do not advocate its use nor did we give a detailed description in the main part of this article, However, it was implemented in [17], so we have included it here to allow the reader a comparison between the previous implementation and ours.

3 Building Blocks for $\mathcal{MQ}$-Signature Cores

Considering Section 2 we see that in order to generate a signature using an $\mathcal{MQ}$-signature scheme we need the following common operations:

- computing affine transformations (*i.e.* vector addition and matrix-vector multiplication),
- (partially) evaluating multivariate polynomials over GF(2^k),
- solving linear systems of equations (LSEs) over GF(2^k).

In this section we describe the main computational building blocks for realizing these operations. Using these generic building blocks we can compose a signature core for any of the presented $\mathcal{MQ}$-schemes (cf Section 4).

3.1 A Systolic Array LSE Solver for GF(2^k)

In 1989, Hochet *et al.* [9] proposed a systolic architecture for Gaussian elimination over GF(p). They considered an architecture of simple processors, used as systolic cells that are connected in a triangular network. They distinguish two different types of cells, main array cells and the boundary cells of the main diagonal.

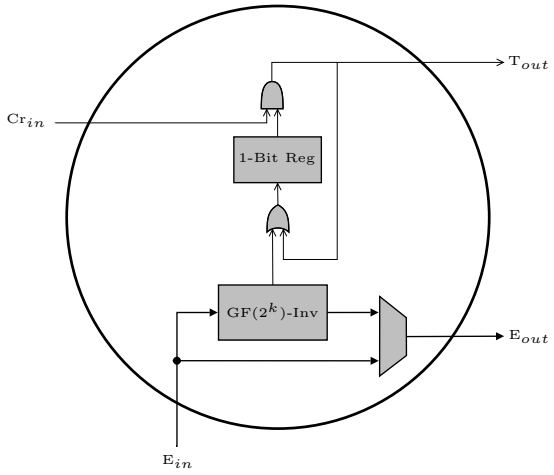

Figure 3: Pivot Cell of the Systolic Array LSE Solver

Wang and Lin followed this approach and proposed an architecture in 1993 [13] for computing inverses over GF(2^k). They provided two methods to efficiently implement the Gauss-Jordan algorithm over GF(2) in hardware. Their first approach was the classical *systolic array* approach similar to the one of Hochet *et al.*. It features a critical path that is independent of the size of the array. A full solution of an $m \times m$ LSE is generated after $4m$ cycles and every m cycles thereafter. The solution is computed in a serial fashion.

The other approach, which we call a *systolic network*, allows signals to propagate through the whole architecture in a single clock cycle. This allows the initial latency to be reduced to $2m$ clock cycles for the first result. Of course the critical path now depends of the size of the whole array, slowing the design down for huge systems of equations. Systolic arrays can be derived from systolic networks by putting delay elements (registers) into the signal paths between the cells.

We followed the approach presented in [13] to build an LSE solver architecture over GF(2^k). The biggest advantage of systolic architectures with regard to our application is the low amount of cells compared to other architectures like SMITH [4]. For solving a $m \times m$ LSE, a systolic array consisting of only m boundary cells and $m(m+1)/2$ main cells is required.

An overview of the architecture is given in Figure 2. The boundary cells shown in Figure 3 mainly comprise one inverter that is needed for pivoting the corresponding line. Furthermore, a single 1-bit register is needed to store whether a pivot was found. The main cells shown in Figure 4 comprise of one GF(2^k) register, a multiplier and an adder over GF(2^k). Furthermore, a few multiplexers are needed. If the row is not initialised yet ($T_{in} = 0$), the entering data is multiplied with the inverse of the pivot (E_{in}) and stored in the cell. If the pivot was zero, the element is simply stored and passed to the next row in the next clock cycle. If the row is initialised ($T_{in} = 1$) the data element $a_{i,j+1}$ of the entering line is reduced with the stored data element and passed to the following row. Hence, one can say that the k-th row of the array performs the k-th iteration of the Gauss-Jordan algorithm.

The inverters of the boundary cells contribute most of the delay time t_{delay} of the systolic network. Instead of introducing a full systolic array, it is already almost as helpful to simply add delay elements only between the rows. This seems to be a good trade-off between delay time and the number of registers used. This approach we call *systolic lines*.

As described earlier, the LSEs we generate are not always solvable. We can easily detect an unsolvable LSE by checking the state of the boundary cells after $3m$ clock cycles (m clock cycles for a systolic network, respectively). If one of them is not set, the system is not solvable and a new LSE needs to be generated. However, as shown in Table 1, this happens very rarely. Hence, the impact on the performance of the implementation is negligible. Table 2 shows implementation

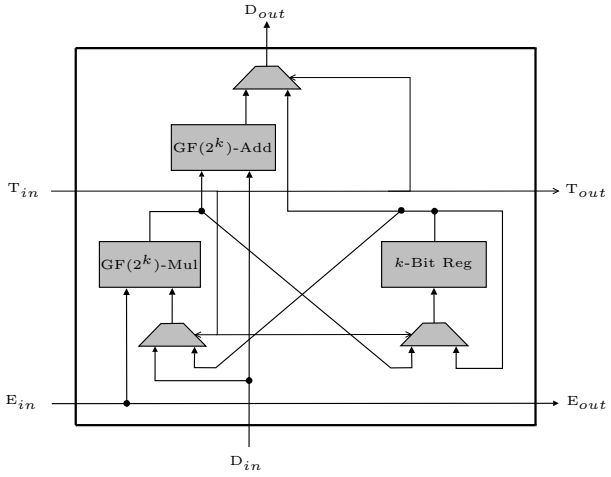

Figure 4: Main Cell of the Systolic Array LSE Solver

results of the different types of systolic arrays for different sizes of LSEs (over GF(2^8)) on different FPGAs.

Table 2: Implementation results for different types of systolic arrays and different sizes of LSEs over GF(2^8) (t_{delay} in ns, F_{Max} in MHz)

	Size on FPGA			Speed		Size on ASIC
Engine	Slices	LUTs	FFs	t_{delay}	F_{Max}	GE (estimated)
Systolic arrays on a Spartan-3 device (XC3S1500, 300 MHz)						
Systolic Array (10x10)	2,533	4,477	1,305	12.5	80	38,407
Systolic Array (12x12)	3,502	6,160	1,868	12.65	79	53,254
Systolic Array (20x20)	8,811	15,127	5,101	11.983	83	133,957
Alternative systolic arrays on a Spartan-3						
Systolic Network (10x10)	2,251	4,379	461	118.473	8.4	30,272
Systolic Lines (12x12)	3,205	6,171	1,279	13.153	75	42,013
Systolic arrays on a Virtex-V device (XC5VLX50-3, 550 MHz)						
Systolic Array (10x10)	1314	3498	1305	4.808	207	36,136
Systolic Lines (12x12)	1,534	5,175	1,272	9.512	105	47,853
Systolic Array (20x20)	4552	12292	5110	4.783	209	129,344

3.2 Matrix-Vector Multiplier and Polynomial Evaluator

For performing matrix-vector multiplication, we use the building block depicted in Figure 5. In the following we call this block a t-MVM. As you can see a t-MVM consists of t multipliers, a tree of adders of depth about $log_2(t)$ to compute the sum of all products $a_i \cdot b_i$, and an extra adder to recursively add up previously computed intermediate values that are stored in a register. Using the `RST`-signal we can initially set the register content to zero.

To compute the matrix-vector product

$$A \cdot b = \begin{bmatrix} a_{1,1} & \dots & a_{1,u} \\ \vdots & & \vdots \\ a_{v,1} & \dots & a_{v,u} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ \vdots \\ b_u \end{bmatrix}$$

using a t-MVM, where t is chosen in a way that it divides[1] u, we proceed row by row as follows:

[1]Note that in the case that t does not divide u we can nevertheless use a t-MVM to compute the matrix-vector product by setting superfluous input signals to zero.

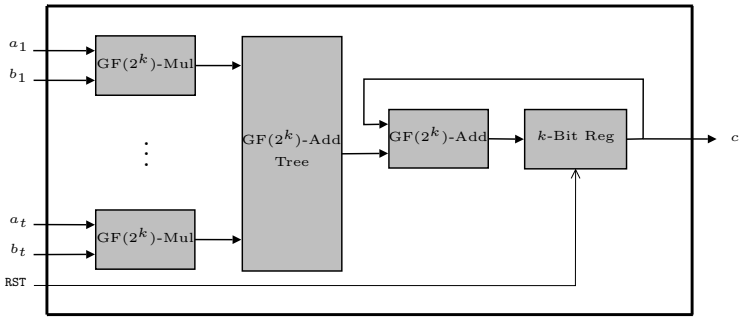

Figure 5: Signature Core Building Block: Combined Matrix-Vector-Multiplier and Polynomial-Evaluator

We set the register content to zero by using `RST`. Then we feed the first t elements of the first row of A into the t-MVM, *i.e.* we set $a_1 = a_{1,1}, \ldots, a_t = a_{1,t}$, as well as the first t elements of the vector b. After the register content is set to $\sum_{i=1}^{t} a_{1,i}b_i$, we feed the next t elements of the row and the next t elements of the vector into the t-MVM. This leads to a register content corresponding to $\sum_{i=1}^{2t} a_{1,i}b_i$. We go on in this way until the last t elements of the row and the vector are processed and the register content equals $\sum_{i=1}^{u} a_{1,i}b_i$. Thus, at this point the data signal c corresponds to the first component of the matrix-vector product. Proceeding in a analogous manner yields the remaining components of the desired vector. Note that the $\frac{u}{t}$ parts of the vector b are *re-used* in a periodic manner as input to the t-MVM. In Section 3.4 we describe a building block, called word rotator, providing these parts in the required order to the t-MVM without re-loading them each time and hence avoid a waste of resources.

Therefore, using a t-MVM (and an additional vector adder) it is clear how to implement the affine transformations $S : \mathbb{F}^n \to \mathbb{F}^n$ and $T : \mathbb{F}^m \to \mathbb{F}^m$ which are important ingredients of an $\mathcal{MQ}$-scheme. Note that the parameter t has a significant influence on the performance of an implementation of such a scheme and is chosen differently for our implementations (as can be seen in Section 4).

Besides realizing the required affine transformations, a t-MVM can be re-used to implement (partial) polynomial evaluation. It is quite obvious that evaluating the polynomials p'_i (belonging to the central map $\mathcal{P}'$ of a $\mathcal{MQ}$-scheme, cf Section 2) with the vinegar variables involves matrix-vector multiplications as the main operations. For instance, consider a fixed polynomial $p'_i(x'_1, \ldots, x'_n) = \sum_{j=1}^{n-m} \sum_{k=j}^{n} \gamma'_{i,j,k} x'_j x'_k$ from the central map of UOV that we evaluate with random values $b_1, \ldots, b_{n-m} \in \mathbb{F}$ for the vinegar variables $x'_1, \ldots, x'_{n-m}$. Here we like to compute the coefficients $\beta_{i,0}, \beta_{i,n-m+1}, \ldots, \beta_{i,n}$ of the linear polynomial

$$p'_i(b_1, \ldots, b_{n-m}, x'_{n-m+1}, \ldots, x'_n) = \beta_{i,0} + \sum_{j=n-m+1}^{n} \beta_{i,j} x'_j \,.$$

We immediately obtain the coefficients of the non-constant part of this linear polynomial, *i.e.* $\beta_{i,n-m+1}, \ldots, \beta_{i,n}$, by computing the following matrix-vector product:

$$\begin{bmatrix} \gamma'_{i,1,n-m+1} & \cdots & \gamma'_{i,n-m,n-m+1} \\ \vdots & & \vdots \\ \gamma'_{i,1,n} & \cdots & \gamma'_{i,n-m,n} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ \vdots \\ b_{n-m} \end{bmatrix} = \begin{bmatrix} \beta_{i,n-m+1} \\ \vdots \\ \beta_{i,n} \end{bmatrix} \tag{1}$$

Also the main step for computing $\beta_{i,0}$ can be written as a matrix-vector product:

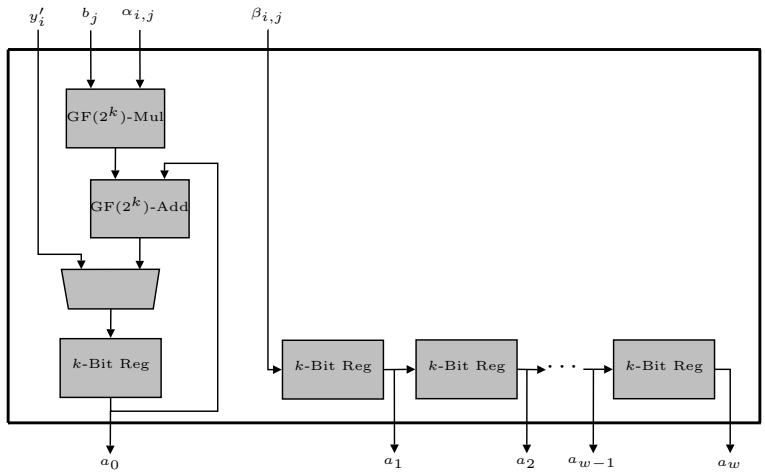

Figure 6: Signature Core Building Block: Equation Register

$$\begin{bmatrix} \gamma'_{i,1,1} & 0 & 0 & \dots & 0 \\ \gamma'_{i,1,2} & \gamma'_{i,2,2} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & & \vdots \\ \gamma'_{i,1,n-m-1} & \gamma'_{i,2,n-m-1} & \dots & \gamma'_{i,n-m-1,n-m-1} & 0 \\ \gamma'_{i,1,n-m} & \gamma'_{i,2,n-m} & \dots & & \gamma'_{i,n-m,n-m} \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ \vdots \\ b_{n-m} \end{bmatrix} = \begin{bmatrix} \alpha_{i,1} \\ \vdots \\ \alpha_{i,n-m} \end{bmatrix} \tag{2}$$

Of course, we can exploit the fact that the above matrix is a lower triangular matrix and we actually do not have to perform a full matrix-vector multiplication. This must simply be taken into account when implementing the control logic of the signature core. In order to obtain $\beta_{i,0}$ from $(\alpha_{i,1} \dots \alpha_{i,n-m})^T$ we have to perform the following additional computation:

$$\beta_{i,0} = \alpha_{i,1} b_1 + \ldots + \alpha_{i,n-m} b_{n-m} \,.$$

This final step is performed by another unit called equation register which is presented in the next section.

3.3 Equation Register

The Equation Register building block is shown in Figure 6. A w-ER essentially consists of $w+1$ register blocks each storing k bits as well as one adder and one multiplier. It is used to temporarily store parts of an linear equation until this equation has been completely generated and can be transferred to the systolic array solver.

For instance, in the case of UOV we consider linear equations of the form

$$p'_i(b_1, \dots, b_{n-m}, x'_{n-m+1}, \dots, x'_n) = y'_i \Leftrightarrow \sum_{j=1}^{n-m} \alpha_{i,j} b_j - y'_i + \sum_{j=n-m+1}^{n} \beta_{i,j} x'_j = 0$$

where we used the notation from Section 3.2. To compute and store the constant part $\sum_{j=1}^{n-m} \alpha_{i,j} b_j - y'_i$ of this equation the left-hand part of an m-ER is used (see Figure 6): The respective register is initially set to y'_i. Then the values $\alpha_{i,j}$ are computed one after another using a t-MVM building block and fed into the multiplier of the ER. The corresponding values b_j are provided by a t-WR building block which is presented in the next section. Using the adder, y'_i and the products can be added up iteratively. The coefficients $\beta_{i,j}$ of the linear equation are also computed consecutively by the t-MVM and fed into the shift-register that is shown on the right-hand side of Figure 6.

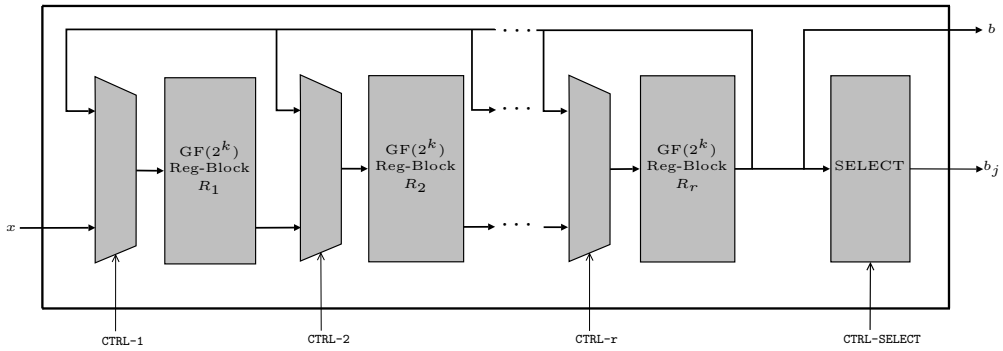

Figure 7: Signature Core Building Block: Word Rotator

3.4 Word Rotator

A word cyclic shift register will in the following be referred to as word rotator (WR). A (t, r)-WR, depicted in Figure 7, consists of r register blocks storing the $\frac{u}{t}$ parts of the vector b involved in the matrix vector products considered in Section 3.2. Each of these r register blocks stores t elements from GF(2^k), hence each register block consists of t k-bit registers. The main task of a (t, r)-WR is to provide the correct parts of the vector b to the t-MVM at all times. The r register blocks can be serially loaded using the input bus x. After loading, the r register blocks are rotated at each clock cycle. The cycle length of the rotation can be modified using the multiplexers by providing appropriate control signals. This is especially helpful for the partial polynomial evaluation where due to the triangularity of the matrix in Equation (2), numerous operations can be saved. Here, the cycle length is $\lceil \frac{j}{t} \rceil$, where j is the index of the processed row. The possibility to adjust the cycle length is also necessary in the case $r > \frac{u}{t}$ frequently appearing if we use the *same* (t, r)-WR, i.e., fixed parameters t and r, to implement the affine transformation T, the polynomial evaluations, and the affine transformation S. Additionally, the WR provides b_j to the ER building block which is needed by the ER at the end of each rotation cycle. Since this b_j value always occurs in the last register block of a cycle, the selector component (right-hand side of Figure 7) can simply load it and provide it to the ER.

4 Performance Estimations of Small-Field $\mathcal{MQ}$-Schemes in Hardware

We implemented the most crucial building blocks of the architecture as described in Section 3 (systolic structures, word rotators, matrix-vector multipliers of different sizes). In this section, the estimations of the hardware performance for the whole architecture are performed based on those implementation results. The power of the approach and the efficiency of $\mathcal{MQ}$-schemes in hardware is demonstrated at the example of UOV, Rainbow, enTTS and amTTS as specified in Section 2.

Side-Note: The volume of data that needs to be imported to the hardware engine for $\mathcal{MQ}$-schemes may seem too high to be realistic in some applications. However, the contents of the matrices and the polynomial coefficients (*i.e.* the private key) does not necessarily have to be imported from the outside world or from a large on-board memory. Instead, they can be generated online in the engine using a cryptographically strong pseudo-random number generator, requiring only a small, cryptographically strong secret, *i.e.* some random bits.

4.1 UOV

We treat two parameter sets for UOV as shown in Table 3: $n = 60$, $n = 20$ (long-message UOV) as well as $n = 30$, $m = 10$ (short-message UOV). In UOV signature generation, there are three basic operations: linearising polynomials, solving the resulting equation system, and an affine transform to obtain the signature. The most time-consuming operation of UOV is the partial evaluation of the polynomials p'_i, since their coefficients are nearly random. However, as already mentioned in the previous section, for some polynomials approximately one half of the coefficients for the polynomials are zero. This somewhat simplifies the task of linearization.

For the linearization of polynomials in the long-message UOV, 40 random bytes are generated to invert the central mapping first. To do this, we use a 20-MVM, a (20,3)-WR, and a 20-ER. For each polynomial one needs about 100 clock cycles (40 clocks to calculate the linear terms and another 60 ones to compute the constants, see (1) and (2)) and obtains a linear equation with 20 variables. As there are 20 polynomials, this yields about 2000 clock cycles to perform this step.

After this, the 20×20 linear system over GF(2^8) is solved using a 20×20 systolic array. The signature is then the result of this operation which is returned after about $4{\times}20{=}80$ clock cycles. Then, the 20-byte solution is concatenated with the randomly generated 40 bytes and the result is passed through the affine transformation, whose major part is a matrix-vector multiplication with a $60{\times}60$-byte matrix. To perform this operations, we re-use the 20-MVM and a (20,3)-WR. This requires about 180 cycles of 20-MVM and 20 bytes of the matrix entries to be input in each cycle.

For the short-message UOV, one has a very similar structure. More precisely, one needs a 10-MVM, a (10,3)-WR, a 10-ER and a $10{\times}10$ systolic array. The design requires approximately 500 cycles for the partial evaluation of the polynomials, about 40 cycles to solve the resulting $10{\times}10$ LSE over GF(2^8) as well as another 90 cycles for the final affine map.

Note that the critical path of the Gaussian elimination engine is much longer than that for the remaining building blocks. So this block represents the performance bottleneck in terms of frequency and hardware complexity. For this reason we decided to clock different components of the design with different frequencies. For the XC5VLX50-3 device the Gaussian elimination engine is clocked with 200 MHz and the rest with 400 MHz. Alternatively, for the XC3S1500 device the Gaussian elimination component is clocked with about 80 MHz, the remaining engines with 160 MHz. See Table 3 for our estimations.

4.2 Rainbow

In the version of Rainbow we consider, the message length is 24 byte. That is, a 24-byte matrix-vector multiplication has to be performed first. One can take a 6-MVM and a (6,7)-WR which require about 96 clock cycles to perform the computation. Then the first 18 variables of x'_i are randomly fixed and 12 first polynomials are partially evaluated. This requires about 864 clock cycles. The results are stored in a 12-ER. After this, the $12{\times}12$ system of linear equations is solved. This requires a $12{\times}12$ systolic array over GF(2^8) which outputs the solution after 48 clock cycles. Then the last 12 polynomials are linearised using the same matrix-vector multiplier and word rotator based on the 18 random values previously chosen and the 12-byte solution. This needs about 1800 clock cycles. This is followed by another run of the $12{\times}12$ systolic array with the same execution time of about 48 clock cycles. At the end, roughly 294 more cycles are spent performing the final affine transform on the 42-byte vector. See Table 3 for some concrete performance figures in this case.

4.3 enTTS and amTTS

Like in Rainbow, for enTTS two vector-matrix multiplications are needed at the beginning and at the end of the operation with 20- and 28-byte vectors each. We take a 10-MVM and a (10,3)-WR for this. The operations require 40 and 84 clock cycles, respectively. One 9-ER is required. Two $10{\times}10$ linear systems over GF(2^8) need to be solved, requiring about 40 clock cycles each. The operation

of calculating the linearization of the polynomials can be significantly optimised compared to the generic UOV or Rainbow (in terms of time) which can drastically reduce the time-area product. This behaviour is due to the special selection of polynomials, where only a small proportion of coefficients is non-zero.

After choosing 7 variables randomly, 10 linear equations have to be generated. For each of these equations, one has to perform only a few multiplications in GF(2^8) which can be done in parallel. This requires about 20 clock cycles. After this, another variable is fixed and a further set of 10 polynomials is partially evaluated. This requires about 20 further cycles.

In amTTS, which is quite similar to enTTS, two affine maps with 24- and 34-byte vectors are performed with a 12-MVM and a (12,3)-WR yielding 48 and 102 clock cycles, respectively. Two 10×10 and one 4×4 linear systems have to be solved requiring for a 10×10 systolic array (twice 40 and once 16 clock cycles). Moreover, a 10-ER is needed. The three steps of the partial evaluation of polynomials requires roughly 40 clock cycles in this case. See Table 3 for our estimations on enTTS and amTTS.

Table 3: Comparison of hardware implementations for ECC and our performance estimations for $\mathcal{MQ}$-schemes based on the implementations of the major building blocks (F=frequency,T=Time, L=luts, S=slices, FF=flip-flops, A=area, XC3=XC3S1500, XC5=XC5VLX50-3)

Implementation	F, MHz	T, μs	S/L/FF	A,kGE	S·T [S·ms]
ECC-163, [1], XC2V200	100	41	-/8,300/1100	-	85.1
ECC-163, CMOS	167	21	-	36	-
ECC-163, [12], XCV200E-7	48	68.9	-/25,763/7,467	-	447.9
UOV(60,20), XC3	80/160	14.625	9821 / 16694 / 5665	149	143.6
UOV(60,20), XC5	200/400	5.85	5334 / 13437 / 5774	143	31.2
UOV(30,10), XC3	80/160	4.188	3060 / 5304 / 1649	46	12.8
UOV(30,10), XC5	200/400	1.675	1585 / 4098 / 1649	43	2.7
Rainbow(42,24), XC3	80/160	7.781	4123 / 7173 / 2332	63	32.1
Rainbow(42,24), XC5	200/400	5.595	2000 / 5626 / 2330	59	11.2
enTTS(28,20), [17], CMOS	80[#]	200	-	22	-
enTTS(28,20), XC3	80/160	2.025	3060 / 5304 / 1649	46	6.2
enTTS(28,20), XC5	200/400	0.81	1585 / 4098 / 1649	43	1.2
amTTS(34,24), XC3	80/160	2.438	3139 / 5434 / 1697	48	7.7
amTTS(34,24), XC5	200/400	0.975	1659 / 4200 / 1697	42	1.6

[#] For comparison purposes we assume that the design can be clocked with up to 80 MHz.

5 Comparison and Conclusions

Our implementation results (as well as the estimations for the optimisations in case of enTTS and amTTS) are compared to the scalar multiplication in the group of points of elliptic curves with field bitlengths in the rage of 160 bit (corresponding to the security level of 2^{80}) over GF(2^k), see Table 3. A good survey on hardware implementations for ECC can be found in [5].

Even the most conservative design, *i.e.* long-message UOV, can outperform some of the most efficient ECC implementations in terms of TA-product on some hardware platforms. More hardware-friendly designs such as the short-message UOV or Rainbow provide a considerable advantage over ECC. The more aggressively designed enTTS and amTTS allow for extremely efficient implementations having a more than 70 or 50 times lower TA-product, respectively. Though the metric we use is not optimal, the results indicate that $\mathcal{MQ}$-schemes perform better than elliptic curves in hardware with respect to the TA-product and are hence an interesting option in cost- or size-sensitive areas.

Acknowledgements. The authors would like to thank our college Christof Paar for fruitful discussions and helpful remarks as well as Sundar Balasubramanian, Harold Carter (University of Cincinnati, USA) and Jintai Ding (University of Cincinnati, USA and Technical University of Darmstadt, Germany) for exchanging some ideas while working on another paper about $\mathcal{MQ}$-schemes.

References

[1] B. Ansari and M. Anwar Hasan. High performance architecture of elliptic curve scalar multiplication. Technical report, CACR, January 2006.

[2] S. Balasubramanian, A. Bogdanov, A. Rupp, J. Ding, and H. W. Carter. Fast multivariate signature generation in hardware: The case of Rainbow. In *ASAP 2008*. to appear.

[3] O. Billet and H. Gilbert. Cryptanalysis of rainbow. In*SCN 2006*, volume 4116 of *LNCS*, pages 336–347. Springer, 2006.

[4] A. Bogdanov, M. Mertens, C. Paar, J. Pelzl, and A. Rupp. A parallel hardware architecture for fast gaussian elimination over GF(2). In *FCCM 2006*, 2006.

[5] G. Meurice de Dormale and J.-J. Quisquater. High-speed hardware implementations of elliptic curve cryptography: A survey. *Journal of Systems Architecture*, 53:72–84, 2007.

[6] J. Ding, L. Hu, B.-Y. Yang, and J.-M. Chen. Note on design criteria for rainbow-type multivariates. Cryptology ePrint Archive `http://eprint.iacr.org`, Report 2006/307, 2006.

[7] J. Ding and D. Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *ACNS 2005*, volume 3531 of *LNCS*, pages 164–175. Springer, 2005.

[8] J. Ding, C. Wolf, and B.-Y. Yang. ℓ-invertible cycles for multivariate quadratic public key cryptography. In *PKC 2007*, volume 4450 of *LNCS*, pages 266–281, Springer, 2007.

[9] B. Hochet, P. Quinton, and Y. Robert. Systolic Gaussian Elimination over GF (p) with Partial Pivoting. *IEEE Transactions on Computers*, 38(9):1321–1324, 1989.

[10] A. Kipnis, J. Patarin, and L. Goubin. Unbalanced Oil and Vinegar signature schemes. In *EUROCRYPT 1999*, volume 1592 of *LNCS*. Springer, 1999.

[11] A. Kipnis, J. Patarin, and L. Goubin. Unbalanced Oil and Vinegar signature schemes — extended version, 2003. 17 pages, `citeseer/231623.html`, 2003-06-11.

[12] C. Shu, K. Gaj, and T. El-Ghazawi. Low latency elliptic curve cryptography accelerators for nist curves on binary fields. In *IEEE FPT'05*, 2005.

[13] C.L. Wang and J.L. Lin. A Systolic Architecture for Computing Inverses and Divisions in Finite Fields $GF(2^m)$. *IEEE TransComp*, 42(9):1141–1146, 1993.

[14] C. Wolf and B. Preneel. Taxonomy of public key schemes based on the problem of multivariate quadratic equations. Cryptology ePrint Archive `http://eprint.iacr.org`, Report 2005/077, 12^{th} of May 2005.

[15] B.-Y. Yang and J.-M. Chen. Rank attacks and defence in Tame-like multivariate PKC's. Cryptology ePrint Archive `http://eprint.iacr.org`, Report 2004/061, 29^{rd} September 2004.

[16] B.-Y. Yang and J.-M. Chen. Building secure tame-like multivariate public-key cryptosystems: The new TTS. In *ACISP 2005*, volume 3574 of *LNCS*, pages 518–531. Springer, July 2005.

[17] B.-Y. Yang, D. C.-M. Cheng, B.-R. Chen, and J.-M. Chen. Implementing minimized multivariate public-key cryptosystems on low-resource embedded systems. In *SPC 2006*, volume 3934 of *LNCS*, pages 73–88. Springer, 2006.