RUHR-UNIVERSITÄT BOCHUM

Implementing Multivariate Quadratic Public Key Signature Schemes on Embedded Devices

Peter Czypek

Diploma Thesis. April 23, 2012.
Chair for Embedded Security – Prof. Dr.-Ing. Christof Paar
Advisor: Stefan Heyse

Abstract

Multivariate quadratic schemes are a promising solution for the need of quantum computer resistant public key signature schemes. However, because this class is relatively young and many schemes of this class were broken in the past, there are only very few implementations available, especially on embedded micro controllers. To estimate if and how these schemes can someday replace current standards it is necessary to know how efficient they can be implemented on various platforms. In this work a theoretical introduction to multivariate quadratic schemes is given. Three schemes which have withstood attacks for some time are then implemented: Unbalanced Oil and Vinegar (UOV), Rainbow and enTTS. Special attention is given to identifying common parts and differences of these schemes, to be able to make basic building blocks for new or modified schemes. A comparison of all three schemes under equal conditions and newest optimizations for three security levels is also performed, which was not available ever before in this form. A summarizing conclusion with a look into future aspects concludes this work.

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Peter Czypek

Contents

1 Introduction

1.1 Motivation

Digital signature schemes have become very important in our daily life. They guarantee message authenticity and integrity, websites use digitally signed certificates to proof their authenticity, video game discs are signed to stop running modified or unlicensed discs and also operating systems check drivers as to whether they contain a digital signature to prevent malicious drivers.
In the future there will be many more applications that rely on digital signatures. But with advancing technology, the attacks on digital signatures might get better, too. One of the most anticipated technologies are quantum computers. With the help of a sufficiently powerful quantum computer it is possible to accelerate a variety of existing algorithms and even develop new ones. This also applies to algorithms which are able to break today's digital signature systems. With the help of the so-called Shor's algorithm [Sho97] which was published in 1995, it is theoretically possible to solve the discrete logarithm problem, on which ECC Systems are based, and the factorization problem which is the basis of RSA. Therefore it is urgently necessary to look for alternatives which withstand even these new threats.
In the past few years many new quantum computer resistant problem classes were found and many new signature and encryption schemes were proposed. For example, code-based systems use the theory of error-correcting codes, lattice-based systems use the problem of finding the shortest vector from one point in space to that lattice, hash-based signature schemes rely on secure hash functions. The class on which the schemes in this work rely is based on Multivariate Quadratic ($\mathcal{MQ}$) polynomials.
It is known that solving systems of $\mathcal{MQ}$-polynomials is very hard, as the corresponding $\mathcal{MQ}$-problem is proven to be $\mathcal{NP}$-complete [GJ79]. This makes the problem suitable for use in cryptographic schemes. But a problem alone is not sufficient. To be able to make the system solvable for the owner of some secret and still not solvable for the rest, it is necessary to embed some hidden trapdoor. Due to the way how the trapdoor is embedded into the systems, another mathematical problem comes into play. In all known schemes the Isomorphism of Polynomials (IP) problem hides the trapdoor. Due to the fact that it is unknown how hard this problem is, almost all $\mathcal{MQ}$ schemes have been broken in the past, including for example the balanced Oil and Vinegar scheme [KS98], Sflash [BFMR11] and much more [Pat95, KS99, GC00, cFJ03, CD03, WBP04]. In summary, nearly all $\mathcal{MQ}$-encryption schemes and most of the $\mathcal{MQ}$-signature schemes have been broken up to this point. There are only very few exceptions like the signature schemes HFE$^-$, Unbalanced Oil and Vinegar (UOV) and its layer based variants Rainbow and enTTS.

UOV resisted all kinds of attacks for 13 years, it is the most promising member of the class of $\mathcal{MQ}$-schemes. If these schemes should become some day a new standard for digital signature schemes, it is important to analyze how and if they can be efficiently implemented, not only on classic PCs but also on constrained devices like on micro controller (μC)s, which are used in a variety of devices, from DVD players to dishwashers.
In this work UOV, Rainbow and enTTS and their latest parameter sets for several security levels are introduced and implemented. Afterwards their runtime, key and code size performances are examined and the schemes are compared to each other. These schemes were chosen as they are quite similar. At the end it is desired to have an overview of how the schemes perform and where the advantages of each scheme are. As the schemes were implemented by the same person we hope to be able to make a fair comparison.
Additionally, this work should give developers who are new to $\mathcal{MQ}$ cryptography a guide on how the single steps in such schemes can be classified and mapped to algorithms which work on memory or arrays.

1.2 Previous and Related Work

There are only a few publications available which deal with the implementation of a $\mathcal{MQ}$ scheme in general. Implementations on micro controllers comparable to that used in this work are not known. Rainbow was implemented in [BCB+08] on a 0.35 μm ASIC, also [TYD+11] implemented Rainbow on an ASIC. Implementations of enTTS were done by Yang *et al.* in [YCCC06] on an ASIC, and in [20004] there was an implementation done on a 8051-compatible μC. The achieved results are compared to the results achieved in this work later in Chapter 4.
This work is based on the newest publications of all schemes, which are for UOV the work of Petzoldt *et al.* [PTBW11], for Rainbow [PBB10b] and for enTTS the publication of Yang *et al.* [YC04].

1.3 Organization

This work starts with an introduction to the theoretical basics which are necessary to understand how $\mathcal{MQ}$ schemes work. First, $\mathcal{MQ}$ schemes in general and later all the schemes are introduced in detail. It is shown how a Public Key System (PKS) can be build with the help of $\mathcal{MQ}$ polynomials and linear maps. Theoretical optimizations are also described and explained. As key generation is an important part of a PKS, it is introduced next.
After explaining the theoretical basics, the schemes are finally implemented. Starting with the used platform and tools, first the general procedure is shown, then the parameter choices are explained and at last the single elements of an implementation

are explained.
Chapter 4 shows actual figures of the achieved run times and also special characteristics are pointed out. A comparison to other schemes and other implementations can also be found in this chapter. A conclusion and an outlook on future aspects concludes this work.

2 Theory

This chapter gives an overview of the mathematical basics which are needed to understand $\mathcal{MQ}$ schemes. First a $\mathcal{MQ}$ scheme is explained in general and then each scheme is introduced in detail. At the end of this chapter an introduction into key generation theory is given.

2.1 Multivariate Qudratic Public Key Systems

This section describes how a $\mathcal{MQ}$ PKS can be build out of multivariate polynomials and linear maps.

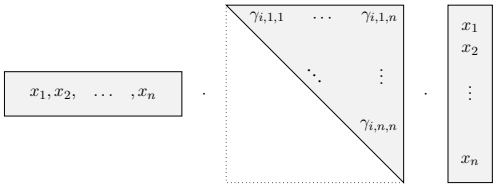
2.1.1 Multivariate Quadratic Polynomials

The basic building blocks for $\mathcal{MQ}$ PKSs are polynomials of a specific form. $\mathcal{MQ}$ polynomials have two properties, they are like the name indicates multivariate which means the polynomials depend on more than one variable and secondly, the polynomial degree is at most quadratic which means only two variables are combined at most. Each variable pair has a coefficient which can also be zero. In general polynomials can contain linear and constant parts. Equation 2.1 shows a general $\mathcal{MQ}$ polynomial.

$$f_i(x_1, ..., x_n) = \sum_{1 \leq j \leq k \leq n} \gamma_{i,j,k} x_j x_k + \sum_{j=1}^{n} \beta_{i,j} x_j + \alpha_i \tag{2.1}$$

The number of variables is denoted by n. Because all combinations of n variables can occur, the maximal number of quadratic monomials is $\frac{n \cdot (n+1)}{2}$, taking into account that $x_i x_j$ is equal to $x_j x_i$. In this work polynomials are visualized by geometric figures to improve the understanding. A general coefficient matrix of a $\mathcal{MQ}$ polynomial has the form of a triangle as seen in Figure 2.1. The figures in this work visualize only the structure of a coefficient matrix, the actual value of a coefficient is not relevant so even a zero coefficient is displayed if it is randomly zero and not by definition. Sometimes the visualization of a single polynomial is not needed. So a set of polynomials is displayed by a rectangle where a single polynomial is represented by a row and the number of polynomials corresponds to the number of columns. Figure 2.2 shows how such a set can be visualized, note that the monomials are in

lexicographical order.



$$\hat{=} f_i(x_1, ..., x_n) = \sum_{1 \leq j \leq k \leq n} \gamma_{i,j,k} x_j x_k$$

Figure 2.1: This figure shows the visualization of quadratic polynomials.

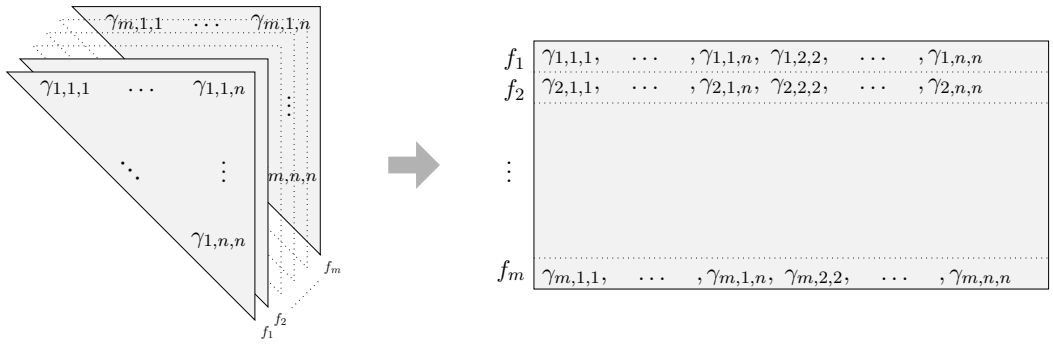


Figure 2.2: Sets of quadratic polynomials can be visualized as a block.

2.1.2 Linear Maps

Linear maps or transformations are also used in $\mathcal{MQ}$ PKS, they are also represented by their coefficient matrices. These maps are always quadratic and have the same dimension as the vector of variables which they are applied to. Linear maps combine variables with each other and therefore hide, mask or mix structures in them. While it is possible to say before the application of a linear map that some variables come from that equation and some from another, afterwards it is not possible any more. Linear maps are invertible if the rank of its coefficient matrix is the maximal rank. This means that all rows and columns have to be linearly independent. This case is very likely in general. In section 3.6.3 self invertible linear maps are introduced which have the properties $A^{-1} = A$ and $A \cdot A = E$. Inversion is then unnecessary.

2.1.3 Public Key System Construction

Having introduced $\mathcal{MQ}$ polynomials and linear maps now it will be shown how a PKS can be build out of these components. The goal of a public key signature

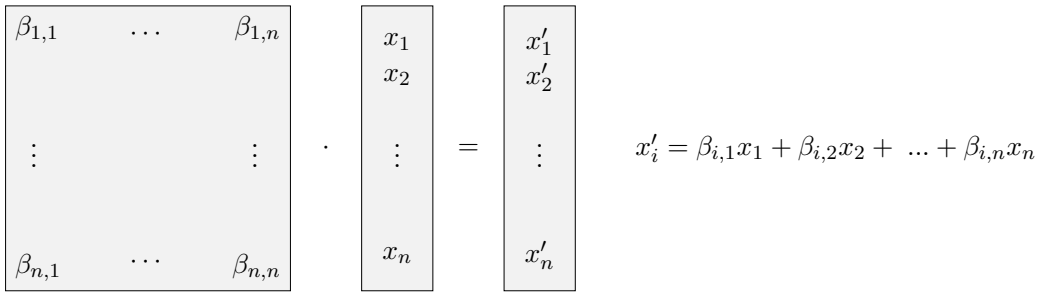


Figure 2.3: Linear maps can be represented by a matrix vector multiplication.

scheme is to generate a public key and secret key pair and a method to sign a message with the private key and verify the signature with the public key. Only the owner of the private key should be able to sign a message. Everybody should be able to verify the signature with the public key and be not able to forge a valid signature for a different message. Furthermore, it should not be possible to derive the private key from the public key in any way.
The following constructions have all $\mathcal{MQ}$PKSs in this work in common, only the number of steps in between or the structure of the maps varies. In a generic $\mathcal{MQ}$ scheme the signature algorithm generates a signature x with a length of n from a message y with a length of m, elements at position i in these are denoted as x_i and y_i. Further, `sign`() is a map from $\mathbb{F}_q^m \to \mathbb{F}_q^n$ and `verify`() is a map from $\mathbb{F}_q^n \to \mathbb{F}_q^m$. The verification works in all schemes in the same way, simply apply the verify algorithm to the signature and check if the output is the message itself. If $\texttt{verify}(\texttt{sign}(y)) = \texttt{verify}(x) = \ y$ holds, then the signature is valid else invalid. Note that the signature algorithm is the inverse of the verify algorithm.

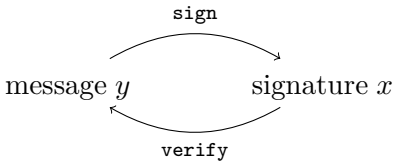


Figure 2.4: The verify algorithm maps the signature to the message and is the inverse of the signature algorithm.

To obtain a well functioning $\mathcal{MQ}$PKS, two conditions must be met: it must be easy to invert the verify map with the knowledge of a secret, and it must be hard to invert the verify map without. The second condition can be met when a large system of multivariate polynomials is chosen and the message should be the solution vector. A public map would then look like

$$\mathcal{P} = \begin{pmatrix} p_1(x_1, \ldots, x_n) \\ \vdots \\ p_m(x_1, \ldots, x_n) \end{pmatrix}$$

and

$$p_k(x_1, \ldots, x_n) := \sum_{1 \leq i \leq j \leq n} \alpha_{k,i,j} x_i x_j = x^{\mathsf{T}} \mathfrak{P}_k x,$$

where $\mathfrak{P}_k$ is the $(n \times n)$ matrix describing the quadratic form of p_k and $x = (x_1, \ldots, x_n)^{\mathsf{T}}$.
The inversion of this system would imply to be able to solve this system of equations in the y variables. And this task is the $\mathcal{MQ}$ problem. To decide if a $\mathcal{MQ}$ sytem is solvable or not is proven to be $\mathcal{NP}$-complete [GJ79]. So how can a trapdoor be included in a large system of polynomials in a way that the first condition is also met? The answer is to generate a combination of small $\mathcal{MQ}$ systems and linear maps which are easily invertible by themselves but hard to invert when combined into one big $\mathcal{MQ}$ map.
The $\mathcal{MQ}$PKS in this work use in general a combination of two linear maps and one central $\mathcal{MQ}$ map. The linear map T is applied to the message, the other map S at last to the variables.

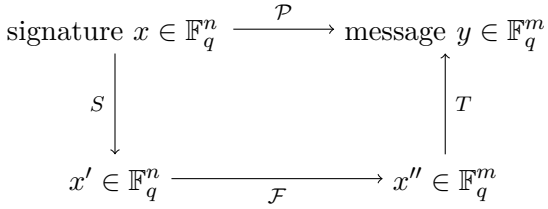


Figure 2.5: This figure shows which maps are part of a $\mathcal{MQ}$-Scheme in general.

Because of the knowledge of S, $\mathcal{F}$ and T and their structure it is possible to invert first $T^{-1}(y) = x''$ then $\mathcal{F}^{-1}(x'') = x'$ and finally $S^{-1}(x') = x$. The composition of these three maps $\mathcal{P} = T \circ \mathcal{F} \circ S$ forms a large unsolvable system. So, in this scenario $S, \mathcal{F}$ and T form the secret key and $\mathcal{P}$ the public key.

2.2 Unbalanced Oil and Vinegar

The name of this scheme comes from the fact that the variables are grouped into two groups, the vinegar and the oil variables, oil and vinegar variables are mixed in the central polynomials, like real oil and vinegar in a salad dressing. The balanced or unbalanced attribute refers to the relation of oil to vinegar variables. In the

unbalanced case there are always more vinegar than oil variables. The Unbalanced Oil and Vinegar scheme is an enhanced version of the broken "balanced" Oil and Vinegar scheme. In a nutshell, in the balanced case it was possible to build separate equations for oil and vinegar parts which made an attack much easier. The unbalanced case which has more vinegar than oil variables was introduced later in [KPG99] and fixed that weakness where a relation of $v = 2 \cdot o$ was proposed to prevent the attack on the balanced $o = v$ case. In section 2.7.1 more details can be found about how the relation can be chosen. Further, n is defined as $n = o + v$. An UOV polynomial has the form in Figure 2.6 and there are $m = o$ polynomials each with $v \cdot o + \frac{v \cdot (v+1)}{2}$ monomials. Equation 2.2 shows the formula for a generic UOV polynomial.

$$f_k(x_1, \ldots, x_n) := \sum_{i \in V, j \in V} \gamma_{k,i,j} x_i x_j + \sum_{i \in V, j \in O} \gamma_{k,i,j} x_i x_j. \tag{2.2}$$

Because the system of equations must be solvable with the message in the result vector, we must choose the number of polynomials equal to the number of the message parts, one polynomial for one message part. Therefore in UOV no oil $\times$ oil terms appear in the polynomials of the central map F. There are initially n variables and o equations, so it is necessary to fix $n - o = v$ variables randomly. After the fixing step, a quadratic $o \times o$ system remains, which can be solved by the Gaussian elimination algorithm. In UOV no transformation on the message parts exists, so the transformation T can be set to the identity matrix or even be left out.

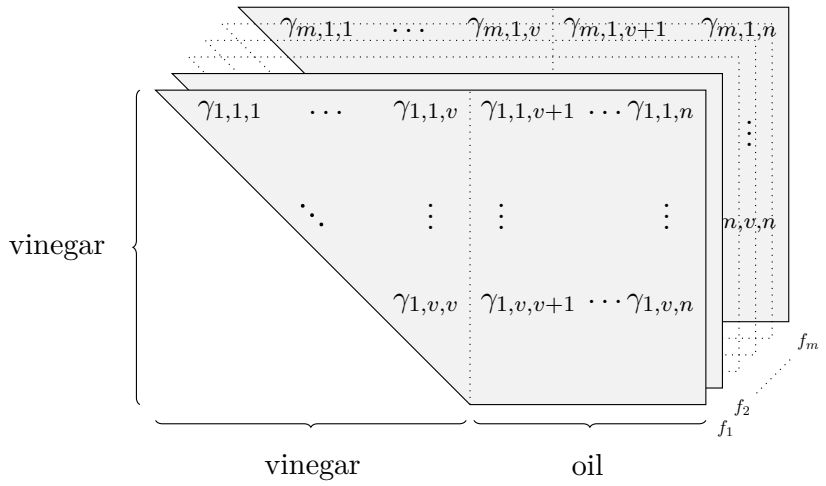


Figure 2.6: Here an Unbalanced Oil and Vinegar central map is shown.

2.3 Rainbow

To minimize the space needed for the private and public coefficients, Rainbow was introduced in [DS05]. Rainbow is a multiple layer variant of UOV. Each layer in Rainbow corresponds to a UOV layer. The layers are not independent from each other. There is a hierarchy which uses the results from the layer above to fix some variables. The number of layers is theoretically not limited, the name comes from the link to the layers of a Rainbow. In the first layer nothing special happens, v variables are fixed randomly and we obtain solutions in $v + o_1$ variables, where o_1 denotes the number of oil variables in the first layer. The next layer uses this solutions of the layer before to fix exactly $v + o_1$ variables in its polynomials so that a $o_2 \times o_2$ system is ready to be solved and to produce a solution for the new o_2 variables. Together with the first layer there is now a solution for $n = v + o_1 + o_2$ variables. This procedure can be applied to all following layers. Figure 2.7 shows how Equation 2.3 and 2.4 look for Rainbow(v, o_1, o_2).

$$f_k(u_1, \ldots, u_n) := \sum_{i \in V_1, j \in V_1} \gamma_{k,i,j} u_i u_j + \sum_{i \in V_1, j \in O_1} \gamma_{k,i,j} u_i u_j \quad \text{for } k = 1, \ldots, o_1 \tag{2.3}$$

$$f_k(u_1, \ldots, u_n) := \sum_{i \in V_1 \cup O_1, j \in V_1 \cup O_1} \gamma_{k,i,j} u_i u_j + \sum_{i \in V_1 \cup O_1, j \in O_2} \gamma_{k,i,j} u_i u_j \quad \text{for } k = o_1 + 1, \ldots, o_1 + o_2 \tag{2.4}$$

Due to the hirarchial structure the parameters of $v, o_1, o_2, \ldots$ can be chosen smaller than in UOV. Smaller parameters lead to a smaller private and public key and also to a smaller message to signature ratio, which is also an advantage of Rainbow.

2.4 Enhanced TTS

Enhanced TTS (enTTS) is as the name indicates an enhanced version of the Tame Triangular System (TTS) scheme. Previous versions of TTS schemes were broken in the past. The current version fixed all weaknesses and has been said to be secure till now. The characteristic of enTTS is that it posses very small private polynomials. Most of the coefficients are zero and no $x_i x_j$ monomial occurs twice, see Table 2.1 for all monomials of enTTS(20,28). Its central map consists of three layers. The first and last layer are normal polynomial systems defined by Equation 2.5 and Equation 2.7, these can be solved by fixing variables and Gaussian elimination. The middle layer defined by Equation 2.6 consists only of two or three equations and only depends on already known variables so that the solution of these layers can be calculated directly.

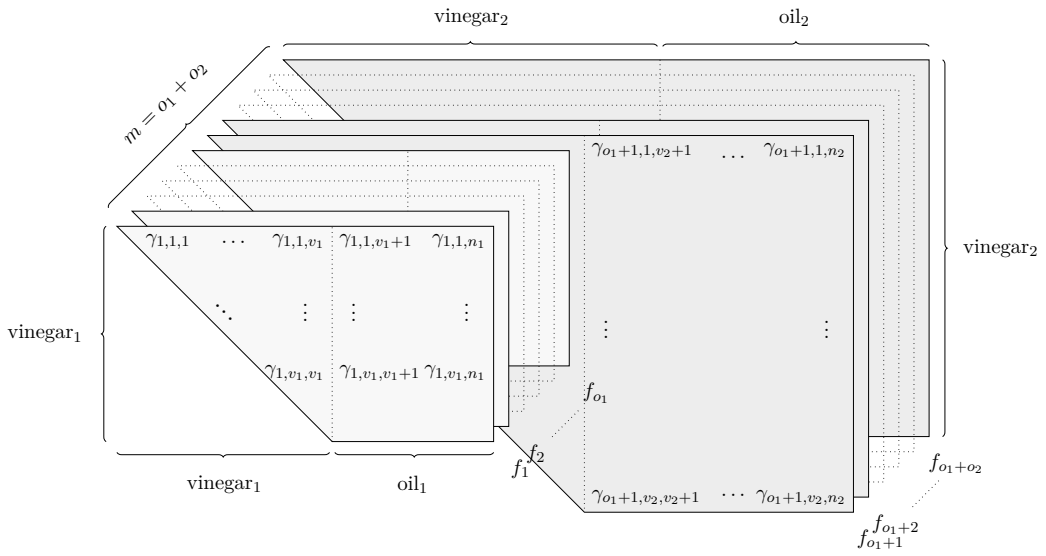


Figure 2.7: The Rainbow central map consists of two or more layers.

$$f_i = x_i + \sum_{j=1}^{2\ell-3} \gamma_{ij} x_j x_{2\ell-2+(i+j+1 \bmod 2\ell-1)} \quad \text{for } 2\ell-2 \le i \le 4\ell-4, \tag{2.5}$$

$$f_i = x_i + \sum_{j=1}^{\ell-2} \gamma_{ij} x_{i+j-(4\ell-3)} x_{i-j-2\ell} + \sum_{j=\ell-1}^{2\ell-3} \gamma_{ij} x_{i+j-3\ell+3} x_{i-j+\ell-2} \tag{2.6}$$
$$\text{for } i = 4\ell-3 \text{ or } 4\ell-2,$$

$$f_i = x_i + \gamma_{i0} x_{i-2\ell+1} x_{i-2\ell-1} + \sum_{j=4\ell-1}^{i} \gamma_{i,j-(4\ell-2)} x_{2(i-j)} x_j \tag{2.7}$$
$$+ \sum_{j=i+1}^{6\ell-3} \gamma_{i,j-(4\ell-2)} x_{4\ell-1+i-j} x_j \quad \text{for } 4\ell-1 \le i \le 6\ell-3$$

Two variations of enTTS were proposed in [YC04], the even and the odd sequence which differ in the number of polynomials and the choice of coefficients. This work focuses only on the odd case, because the odd case is comparable with the implementation of enTTS(20,28) in [YCCC06]. The extreme small private key makes enTTS very efficient in terms of private key size and signature generation time, compared to UOV and Rainbow. By comparison of the polynomials, enTTS can be seen as a special case of Rainbow with three layers and very sparse polynomials. Due to the parameters the public key is bigger compared to the other systems.
Note that the formula differs from the one in the original paper due to a small error

Table 2.1: This table shows all quadratic monomials of the central map of enTTS(20,28). The number denotes the polynomial in which the coefficient occurs.

f_i	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}	x_{20}	x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}
x_0																				19	20	21	22	23	24	25	26	27
x_1							17		16	8	9	10	11	12	13	14	15											
x_2						17		18	15	16	8	9	10	11	12	13	14			20	21	22	23	24	25	26	27	
x_3					17		18		14	15	16	8	9	10	11	12	13											
x_4						18			13	14	15	16	8	9	10	11	12			21	22	23	24	25	26	27		
x_5									12	13	14	15	16	8	9	10	11											
x_6									11	12	13	14	15	16	8	9	10			22	23	24	25	26	27			
x_7									10	11	12	13	14	15	16	8	9											
x_8											19									23	24	25	26	27				
x_9												20					17											
x_{10}													21			17		18		24	25	26	27					
x_{11}														22	17		18											19
x_{12}														17	23	18				25	26	27					19	20
x_{13}															18	24										19	20	21
x_{14}																	25			26	27				19	20	21	22
x_{15}																		26						19	20	21	22	23
x_{16}																			27	27			19	20	21	22	23	24
x_{17}																						19	20	21	22	23	24	25
x_{18}																					19	20	21	22	23	24	25	26

in the second sum of the middle layer. The formula was corrected and verified to generate the example sequences given in the original work. The author was contacted, but did not respond to this issue.

2.5 Compressed and 0/1 UOV

These modifications on UOV were developed by Petzold *et al.* in [PTBW11] with the goal of minimizing the public key size. The core idea of this modification is to define a large part of the public key cyclic as proposed in [PBB10a] or even randomly fixed. To be able to generate a structure in the public key the authors define an inverse way of key generation which is described in detail in the next section.
The idea behind 0/1 UOV is to use only elements from $\mathbb{F}_2$ in the fixed public key part. Theoretically by this approach the key size shrinks even more because eight $\mathbb{F}_2$ elements can be saved in a byte. Due to simpler arithmetic in $\mathbb{F}_2$ even the needed verification time decreases.
Additionally, further security measures must be taken to guarantee that other attacks are not possible. A key which only consists of variables in $\mathbb{F}_2$ is easier to attack than one consisting only of variables from $\mathbb{F}_{2^8}$.
To maintain the security level it is necessary to make sets in which every node is con-

nected to a number of other nodes in a way that even when a number of variables is fixed the unknown variables are not reduced to only $\mathbb{F}_2$ elements. The optimal solution is to choose the oil × oil part monomial ordering, which is the security relevant part in the public key, according to a complementary Turań graph, like shown in [PTBW11][Section 5.3]. Such a graph fulfils these requirements with a minimal number of edges. In Section 3.3.3 an algorithm for generating such a graph can be found.

2.6 Key Generation

As already mentioned the private maps form the private key and their composition the public key. To compute the composition there exist two ways.
The first one requires multiplication of S with $\mathcal{F}$ in quadratic form. Equation 2.8 shows the formula for generating $\mathcal{P}$ from $S, \mathcal{F}$ and T. It shows how the linear map S "mixes" the variables in each polynomial and how the t_{ij} from matrix T "mix" the polynomials with each other.

$$\mathfrak{P}^{(i)} = \sum_{j=1}^{m} t_{ij} S^{\intercal} \mathfrak{F}_j S \tag{2.8}$$

The second way is based on the idea of transforming the multiplications $S^{\intercal}\mathfrak{F}^{(j)}S$ to a multiplication with a single matrix A: $\mathcal{P} = \mathcal{F} \cdot A$. Note that one polynomial from $\mathcal{F}$ is multiplied by a row of A, elements are ordered lexicographically in $\mathcal{F}$. Matrix A is in [PTBW11] defined like in Equation 2.9.

$$a_{ij}^{rc} = \begin{cases} s_{ri} \cdot s_{ci}, & (i = j) \\ s_{ri} \cdot s_{cj} + s_{rj} \cdot s_{ci}, & \text{otherwise} \end{cases} \tag{2.9}$$

The element a_{ij}^{rc} describes the element indices quadratically, it can be also written as $a_{[(i \cdot n - \frac{i(i+1)}{2} + j],[(r \cdot n - \frac{r(r+1)}{2}) + c]}$ with n the number of UOV variables.
This approach also allows the inverse way, defining first matrix B which is the upper $v \times n$ matrix of $\mathcal{P}$ and then calculating $\mathcal{F}$ by inverting matrix A and calculating $\mathcal{F} = B \cdot A^{-1}$. The structure of matrix B can be chosen freely, it only has to be uniformly random and have full rank. The full rank property is given when the left $o \times o$ matrix is defined as the identity matrix. The structure of B is also transferred to $\mathcal{P}$.
The size of the keys corresponds to the size of all coefficients. The number of elements in $\mathcal{P}$ is in all schemes $m \cdot (\frac{n(n+1)}{2})$, but in the case of Compressed UOV or 0/1 UOV, as mentioned before, large parts can be chosen freely and are redundant, so that they can be treated as a system parameter or can be generated.

2.7 Attacks

In this section a short overview of attacks on $\mathcal{MQ}$ schemes is given. Actual security parameters are derived from these attacks later in Section 3.3.2.

2.7.1 UOV

Direct Attack.

To forge a single signature an attacker would have to solve a system of o quadratic equations in v variables over $\mathbb{F}_q$. The usual way of finding one solution starts with guessing v variables at random. This preserves one solution with high probability. The best way of solving the remaining $\mathcal{MQ}$-system of o equations and variables is to guess a few further variables and then apply a Gröbner Basis algorithm like F_4 (see Hybrid Approach of Bettale *et al.* [BFP09]). Recently Thomae *et al.* showed that it is more efficient than guessing v variables at random [TW12]. Calculating these v variables through linear systems of equations allows to solve a system of $o - \lfloor \frac{v}{o} \rfloor$ quadratic equations and variables afterwards. To determine the complexity of solving a $\mathcal{MQ}$-system using a Gröbner basis algorithm like F_4 the work of Bettale *et al.* [BFP09] provides detailed information. In a nutshell, first the degree of regularity d_{reg} has to be calculated. For semi-regular sequences, which generic systems are assumed to be, the degree of regularity is the index of the first non-positive coefficient in the Hilbert series $S_{m,n}$ with

$$S_{m,n} = \frac{\prod_{i=1}^{m}(1 - z^{d_i})}{(1-z)^n},$$

where d_i is the degree of the i-th equation. Then, the complexity of solving a zero-dimensional (semi-regular) system using F_4 [BFP09, Prop. 2.2] is

$$\mathcal{O}\left(\left(m\binom{n + d_{reg} - 1}{d_{reg}}\right)^{\alpha}\right),$$

with $2 \leq \alpha \leq 3$ the linear algebra constant. In this work $\alpha = 2$ was used.

Key Recovery Attacks.

There are two key recovery attacks known so far. The first is a purely algebraic attack called *Reconciliation* attack [BD08]. In order to obtain the secret key S, $\binom{k+1}{2}o$ quadratic equations must be solved in kv variables for an optimal parameter $k \in \mathbb{N}$. The second attack is a variant of the *Kipnis-Shamir* attack on the balanced Oil and Vinegar scheme [KS98]. The overall complexity of this attack is $\mathcal{O}(q^{v-o-1}o^4)$. Note that $v = 2o$ is very conservative in order to prevent this attack and thus v can be chosen much smaller for o sufficiently large.

2.7.2 Rainbow

All attacks against UOV also apply to Rainbow. Additionally, the security of Rainbow relies on the MinRank-problem. Therefore, both MinRank and HighRank attacks must be taken into account. Petzold *et al.* provide in [PBB10b] a comprehensive security analysis, therefore we reference to this work for an overview of attacks and parameters.

2.7.3 enTTS

All attacks against Rainbow also apply to enTTS. The only attack that benefits from the differences between Rainbow and enTTS is the Reconciliation attack with large k. But as the complexity of this attack is out of reach, this does not affect the security. In fact, the complexity is higher than the ones of all the other attacks, so it can be omitted when choosing parameters. More important is the slight benefit of the Band Separation attack. For the odd sequence enTTS $m + n - 1$ quadratic equations in $n - 2$ instead of n variables can be derived.

3 Implementation

The main focus of this work is on the implementation. In the following sections the implementation is described together with all implementation relevant information, like for example how the field and the parameters were chosen.

3.1 Platform and Tools

An ATxMega128a1 [Atmb] on a xplain [Atmc] board was used as target device. This micro processor has a clock frequency of 32 MHz, 128KB flash program memory and 8KB SRAM. The code was written in C and optimized for embedded use. As compiler avr-gcc in version 4.5.1 and at some places assembler gcc-as 2.20.1 was used. The controller is connected to a PC via a USB cable. There is additionally an AT90USB1287 [Atma] controller on the xlain board for an UART to USB bridge, it is flashed with LUFA firmware [Pro] which can also be configured to act as an on device programmer for the ATxMega128a1.

3.2 General Procedure

In this section the general procedures are described. Including the way how signature, verification and key generation algorithms work and how they were embedded in a real signature generation and verification application. To verify that the implementation is correct both algorithms were flashed to the controller if enough space was available for both keys.

3.2.1 Key Generation

The key generation algorithm was implemented in C on a PC. When starting the key generation executables, they produce a key pair and a header file containing the used parameters for the algorithms and symbols for the linker. These three generated files should be placed in the algorithm directory and included in the algorithm project. The algorithm is implemented such that it awaits the parameters in a header file as constant definitions (`#define constantname value`) and the keys in the key files. The symbols in the header file are linked to the keys after compilation by the linker.

3.2.2 Sign

At the begin of each function the random number generator must be seeded. This happens in the `init()` function of each scheme, no further initialization is required. After this initialization the UART interface is observed for incoming bytes. The `sign()` function awaits only one parameter, the message, and starts when enough bytes have been received. When the signature is calculated it is send back to the PC via the UART interface. How the signature process looks in detail can be found in the corresponding sections for UOV, Rainbow, and enTTS.

3.2.3 Verify

The verification procedure works in the same way as the signature with the only difference that it awaits two parameters, the message and the signature. When the verification is complete it only outputs a text message which says whether or not the signature is valid. Both parameters can be sent serially one after another or concatenated as one value to the μC.
Verification looks all the same for all schemes, except that enTTS verifies additionally the linear parts. Basically verification is only a polynomial evaluation as it takes place when the vinegar variables are fixed, as described later in Section 3.8.

3.3 Choice of Parameters

When choosing parameters for a cryptographic scheme two main aspects must be taken into account. These are the given security for a parameter set and its potential to be implemented efficiently. MQ schemes are defined very generally over a field where multiplication and addition are defined. Also number and relation of variables are defined as generic values like o,v or l, because the schemes should be easily parametrizable. For example if a rise of parameters due to a new attack is necessary, an adjustment should require little effort. In the case of UOV for example the parameters were risen a couple of times in the past.

3.3.1 Field

Regarding the used field, there are not many alternatives on a μC. As the implementation should be fast the natural choice of the field on an 8 bit μC is $\mathbb{F}_{2^8}$. As we can address and progress a byte the best, addition is a xor operation and multiplication can be done with modular addition in the exponent. Choosing multiples of a byte like $\mathbb{F}_{2^{16}}$ or fractions like $\mathbb{F}_{2^4}$ would require a sort of element splitting in two bytes or a merge of two bytes every time an element is read out or processed. This would increase the runtime extremely. Odd fields would require even more

complicated arithmetic. Whether or not the gain of security and maybe smaller key sizes by choosing a larger field could compensate the increased runtime remains an open question.

3.3.2 Algorithm Paramters

According to the attacks mentioned in Section 2.7, Table 3.1 was filled with the minimal required parameters for each security level. If attacks do not apply to a scheme or play no role, the values are left out.

Table 3.1: Minimal parameters for UOV, Rainbow and enTTS achieving certain levels of security. Thereby g is the optimal number of variables to guess in the hybrid approach and k is the optimal parameter selectable for the Reconciliation attack.

Scheme	Security	Parameter	Direct attack	Band Separation	MinRank	HighRank	Kipnis-Shamir	Reconciliation
UOV (o, v)	2^{64}	$(21, 28)$	2^{67} $(g = 1)$	-	-	-	2^{66}	2^{131} $(k = 2)$
	2^{80}	$(28, 37)$	2^{85} $(g = 1)$	-	-	-	2^{83}	2^{166} $(k = 2)$
	2^{128}	$(44, 59)$	2^{130} $(g = 1)$	-	-	-	2^{134}	2^{256} $(k = 2)$
Rainbow (v, o_1, o_2)	2^{64}	$(15, 10, 10)$	2^{67} $(g = 1)$	2^{70}	2^{141}	2^{93}	2^{125}	2^{242} $(k = 6)$
	2^{80}	$(18, 13, 14)$	2^{85} $(g = 1)$	2^{81}	2^{167}	2^{126}	2^{143}	2^{254} $(k = 5)$
	2^{128}	$(36, 21, 22)$	2^{131} $(g = 2)$	2^{131}	2^{313}	2^{192}	2^{290}	2^{523} $(k = 7)$
enTTS (ℓ, m, n)	2^{64}	$(7, 28, 40)$	2^{89} $(g = 1)$	2^{68}	2^{126}	2^{117}	2^{127}	-
	2^{80}	$(9, 36, 52)$	2^{110} $(g = 2)$	2^{85}	2^{159}	2^{151}	2^{160}	-
	2^{128}	$(15, 60, 88)$	2^{176} $(g = 3)$	2^{131}	2^{258}	2^{249}	2^{259}	-

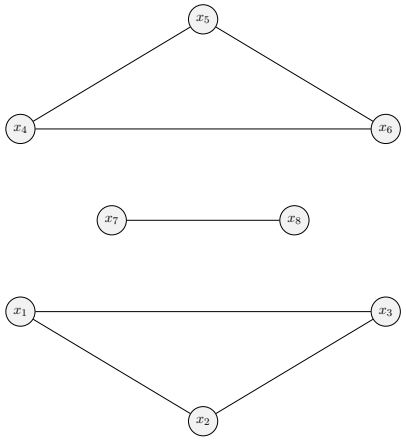
3.3.3 Turan graph for 0/1 UOV

The 0/1 UOV scheme requires the use of an inverse Turań graph to change the ordering of polynomials as mentioned in Section 2.5, so it is necessary to construct such a graph. As this happens on a PC the process is not optimized for speed or other points.
A Complementary Turań graph $CT(n, k)$ has the property that it does not contain a $(k + 1)$ independent set and has the minimal number of edges, which is bounded from below by $\lceil \frac{n}{2} \cdot (\frac{n}{k} - 1) \rceil$ [PTBW11]. Further the graph is divided into k sets: n mod k sets with $\lceil \frac{n}{k} \rceil$ nodes and $(k - n)$ mod k sets with $\lfloor \frac{n}{k} \rfloor$ nodes.
The algorithm must calculate the number of sets and their sizes first, then it must form the sets by filling the first groups with nodes which are not in a set already until the first set is filled. When a set is filled the algorithm continues with the next available node. With the same method the second, smaller sets are filled. An

example graph can be found in Figure 3.1 which was generated by Algorithm 3.3.1.

Algorithm 3.3.1: Turań Graph Generation

Input : Number of variables n, Number of oil variables o
Output : Turań graph
begin
  while *number of edges* $> (o * (o + 1))/2$
  do
    `/* calculate minimal number of edges for actual` k `*/`
    number of edges $= \lceil \frac{n}{2} \cdot (\frac{n}{k} - 1) \rceil$
    `/* increase number of groups` k `*/`
    $k++$
  `/* calculate number of sets and corresponding sizes */`
  num1 $= n \bmod k$
  num2 $= k - (n \bmod k)$
  size1 $= \lceil \frac{n}{k} \rceil$
  size2 $= \lfloor \frac{n}{k} \rfloor$
  `/* fill sets with nodes */`
  for *sets with size1* **do**
    put next free node in current set
  for *sets with size2* **do**
    put next free node in current set



$$\ldots + x_1x_2 + x_1x_3 + x_2x_3 + x_4x_5 + x_4x_6 + x_5x_8 + x_7x_8$$

Figure 3.1: In this figure CT(8,3) and its polynomial representation is shown.

3.4 Polynomial Representation / Key Storage

When implementing $\mathcal{MQ}$PKS on microprocessors it is important to construct an efficient way of storing and reading the keys out of memory. All polynomials of an $\mathcal{MQ}$-scheme are represented by their coefficients. It is important to decide how this coefficients are processed during runtime. The coefficients of UOV and Rainbow can be easily mapped to some readout loops. This is not easy with enTTS as only a minimal count of coefficients are used and these few coefficients are spread over three layers and six different cyclic structures. As random access on the flash memory produces a lot of addressing overhead while calculating the address each time, here a serial approach was chosen. All coefficients are stored in memory in the same exact order in which they are read out. There are no gaps or zeros in memory which is also memory efficient. Figure 3.2 illustrates the memory structure. This memory structure allows to read out the keys directly and simply increment the address to reach the next coefficient. The AVR instruction set allows a memory readout with a post increment in one clock cycle from SRAM or two clock cycles from Flash memory.
Therefore, no additional address calculation is needed. The number of coefficients to store and thus the memory consumptions in bytes is $o\left(ov + \frac{v(v+1)}{2}\right)$ for UOV, $o_1\left(o_1 v + \frac{v(v+1)}{2}\right) + o_2\left(o_2(v+o_1) + \frac{(v+o_1)(v+o_1+1)}{2}\right)$ for Rainbow and $8l^2 - 6l - 3$ for enTTS. The resulting memory requirements for specific security parameters are given in Table 3.2.

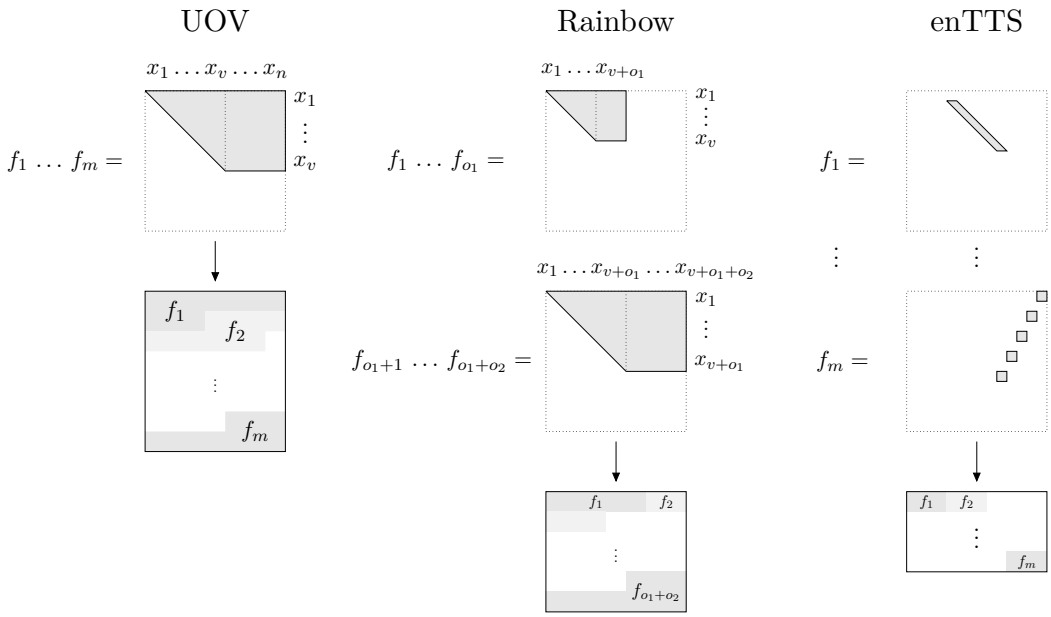


Figure 3.2: This figure shows the central map coefficient to memory mapping.

3.5 Arithmetic

The schemes are defined over a Galois field. In this mathematical construct, addition, subtraction, multiplication and division are redefined. The field used in this work is $GF(2^8)$ which contains 255 elements. These 255 elements can be seen as polynomials with eight coefficients over F_2. In an eight bit value every bit represents one coefficient. For example the byte `0x11` represents the polynomial $0 \cdot x^7 + 0 \cdot x^6 + 0 \cdot x^5 + 1 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^2 + 0 \cdot x^1 + 1 \cdot x^0 = x^4 + 1$. The most significant bit represents the coefficient of the highest exponent.
To construct the whole Galois field with all 255 elements, a generator polynomial must be chosen. In this work the polynomial `0x11D` $= x^8 + x^4 + x^3 + x^2 + 1 = 0$ is used. In general in $GF(p^m)$ there exist $\frac{\varphi(p^m-1)}{m}$ primitive polynomials, which is equal to 16 for the parameters used here. Non-primitive polynomials have a lower rank and do not generate all elements. With this generator all elements are obtained by multiplication of every element by x over and over again, e.g. first element $e_1 = 1 =$ `0x01`, second element $e_2 = e_1 \cdot x = 1 \cdot x = x =$ `0x02` and so on. Note that a multiplication by x is an integer multiplication by 2, which again can be represented by a left shift.
In the case of an overflow, e.g. in $e_8 = x^7, e_9 = e_8 \cdot x = x^8$, the generator polynomial is used to reform the equation. With the generator equation, which is represented by `0x11D` = `100011101` $= x^8 + x^4 + x^3 + x^2 + 1 = 0$ and with the subtraction of x^8 on both sides, the value x^8 can be written as $x^4 + x^3 + x^2 + 1$, therefore e_9 is `00011101` = `0x1D`, which now fits into an eight bit value. By looking at the bit representation, this step can be implemented by a xor operation with the generator polynomial; every time an eight bit overflow occurs in the left shift, the overflowing one is eliminated by the leading one of the generator polynomial. By repeating these shift-check-xor steps until the first element is reached again, all field elements can be computed.
The addition and also the subtraction in this field is the same and is a simple xor operation, because the coefficients are in $\mathbb{F}_2$. Multiplication and division can be performed by classical polynomial multiplication followed by reduction or by using a table lookup method. In this work the table lookup method is chosen. The element table also represents the exponentiation table because exponentiation consists of multiple multiplications with the same element and this is how the table is generated. By reordering the upper table by its times of multiplication, a logarithm table of all elements is constructed. Because the Galois field is a ring this does not only apply to one element but to all elements. Now a product of two elements can be computed by transforming them into exponential representation, summing them up and transforming the result back to the normal representation by using the logarithm table, $e_1 \cdot e_2 = log(exp(e_1) + exp(e_2) \bmod 255)$. The same applies to division but with subtraction, $\frac{e_1}{e_2} = log(exp(e_1) - exp(e_2) \bmod 255)$. The used plus and minus operators are not the field operators, but standard integer operators, so it must be checked if the result is still in the field. If the logarithm argument is greater than 255 (for multiplication) or less than 0 (in case of division), 255 must be

added or subtracted. A modulo operation with 255 would have the same effect but would cost more clock cycles. This simple checking is legit because the argument can grow at most to $2 \cdot 255$ which comes back into the range by subtraction of 255. The routine with the range check needs 42 clock cycles, the same routine with a modulo operation would need 275 clock cycles, around five times more.
The exponentiation and logarithm tables are stored in the flash, in the declaration they are marked with the keyword `PROGMEM`, to avoid copying to SRAM at startup, which is the gcc default behaviour.
The tables occupy $2 \cdot 256$ Bytes = 512 Bytes of flash memory. The functions for multiplication and division are `mul(a,b)` and `div(a,b)` and expect two elements as input parameters and return the multiplication result as an eight bit value. The division function has the same form and parameters. For addition and subtraction, no extra function but the standard C xor operator `^` is used. Additionally versions of `mul(a,b)` and `div(a,b)` have been implemented which return the result in exponential form instead of transforming the result back to the normal representation. These functions are used to minimize memory access as described in Section 3.6.1.

3.6 Optimizations

3.6.1 Calculation in Exponential Representation

Our multiplication and division utilizes table look-ups to compute the result. Often the back transformation is not necessary. For example one way to multiply three values is to multiply the first two by a look-up followed by an addition and a reverse look-up. With the temporary result and the third factor the same would be done.
The second way is to keep the temporary result in exponential form and not transform it back and forth. This would save unnecessary memory access. Therefore, temporary results in all algorithms are kept as long as possible in this representation. Because all coefficients of the keys are first used in a multiplication, the transformation to the logarithmic presentation can be accomplished at key generation time. This applies to the private and public map as well as to the linear maps.

3.6.2 Reduced Polynomials

Not all possible monomials add to security in the $\mathcal{MQ}$ schemes. It is possible to remove some of them without decreasing security. The same applies to constant terms in the linear maps. The removal beneficially affects the required key space and runtime.
In the central UOV and Rainbow polynomials all linear and constant parts were removed. Also the linear maps contain no constant terms in this implementation. In enTTS the central polynomials were left untouched and only the linear maps were modified, because the security analysis of enTTS is more complex than of the other

ones.

3.6.3 Choosing a Self-Invertible S

According to [PTBW11] so-called equivalent keys exist. By picking any representative of a key, the structure of the central equations remains valid. By looking at the way how the linear map is used in the UOV scheme, large parts can be chosen more efficiently without affecting the structure of the central map $\mathcal{F}$, for details see [PTBW11][Chapter 4 Security of UOV]. For a key recovery attack it is sufficient to find any representation of the key so this assumption is allowed. The same form can be applied to the Rainbow linear map. In Section 3.9 it is shown how the number of multiplications during signature generation can be reduced using a S matrix in this form.

Another side effect of choosing S of this form is that in UOV not only parts of $\mathcal{P}$

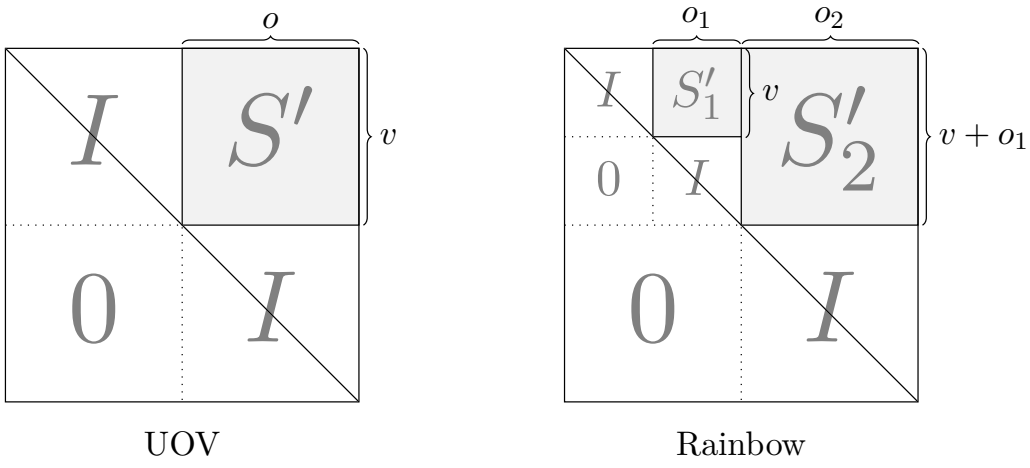


Figure 3.3: This figure shows the structure of S when the theory of equivalent keys is used.

can be treated as a system parameter but also a part of $\mathcal{F}$. Because of the identity matrix in the $v \times v$ part in S this part is the same in $\mathcal{P}$ and $\mathcal{F}$. In a system in which both, sign and verify, are required, this parameter could be saved only once.

3.6.4 Precalculation of Quadratic Variables

In $\mathcal{MQ}$ schemes usually two variables are fixed and then multiplied by a third coefficient or variable. The first two fixed variables are fixed equally for all polynomials. It is much more efficient to save these quadratic combinations of variables and reuse them every time they must be multiplied by a new coefficient. This technique can be applied to all polynomial evaluation steps. These occur in the sign algorithm

in the vinegar fixing and evaluation part and also in the verify algorithm when the signature variables are multiplied by the public key. The savings are $(m-1) \cdot n \cdot \frac{n+1}{2}$ multiplications in the verification step and $(m-1) \cdot v_f \cdot \frac{v_f+1}{2}$ in the signing step, where v_f denotes the number of variables which are fixed.

3.7 Random Numbers

For random numbers, a pseudo-random number generator is used. The used seed is taken from the whole uninitialised SRAM memory. This exploits the fact that some SRAM cells flip randomly into a charged (one) or an uncharged state (zero) at start up. All these values are summed up to one value. When calling `rand()` a pseudo-random eight bit value is returned. Random numbers are used to fix the vinegar variables randomly and for key generation. The code for generating the seed can be found in the appendix in Listing A.1.

3.8 Inverting the Central Maps

Inverting the central maps is the biggest part in inverting the private maps. In this section it is showed how the central maps can be inverted and what must be considered in an implementation.

3.8.1 UOV and 0/1 UOV

By choosing random vinegar variables and evaluating the polynomials, the equations become much smaller. In the oil and vinegar polynomials there are some coefficients which are multiplied twice by the vinegar variables and some which are only once. The coefficients for which two multiplications occur turn into $\frac{v \cdot (v+1)}{2}$ constant values for each polynomial. Note that quadratic vinegar $\times$ vinegar terms are only computed once and then used for all following equations as described in Section 3.6.4. The remaining $o \cdot v \cdot m$ values are the product of one coefficient from the central polynomial, one oil variable and one vinegar variable and become therefore linear in the oil variables. All these values are connected with an addition operation which equals a xor operation in $\mathbb{F}_{2^8}$. Therefore, all constant terms are added up to one value per polynomial and the remaining oil terms into o linear terms, as shown in Equations 3.1 to 3.3 for UOV(4,2).

$$x_1 \cdots x_4 \leftarrow \text{fixed random values } r_1 \cdots r_4$$

$$\begin{aligned} f_i \;= \gamma_{i,1,1}r_1r_1 + \gamma_{i,1,2}r_1r_2 + \gamma_{i,1,3}r_1r_3 + \gamma_{i,1,4}r_1r_4 \;&+\gamma_{i,1,5}r_1x_5 + \gamma_{i,1,6}r_1x_6 \\ +\gamma_{i,2,2}r_2r_2 + \gamma_{i,2,3}r_2r_3 + \gamma_{i,2,4}r_2r_4 \;&+\gamma_{i,2,5}r_2x_5 + \gamma_{i,2,6}r_2x_6 \\ +\gamma_{i,3,3}r_3r_3 + \gamma_{i,3,4}r_3r_4 \;&+\gamma_{i,3,5}r_3x_5 + \gamma_{i,3,6}r_3x_6 \\ \underbrace{+\gamma_{i,4,4}r_4r_4}_{\sum \text{constant terms} = v_i} \;&\underbrace{+\gamma_{i,4,5}r_4x_5 + \gamma_{i,4,6}r_4x_6}_{\sum \text{linear terms} = c_{i,5}x_5, c_{i,6}x6} \end{aligned} \tag{3.1}$$

$$\begin{aligned} v_i \;&=\; \gamma_{i,1,1}r_1r_1 + \gamma_{i,1,2}r_1r_2 + \gamma_{i,1,3}r_1r_3 + \gamma_{i,1,4}r_1r_4 + \gamma_{i,2,2}r_2r_2 \\ &+\; \gamma_{i,2,3}r_2r_3 + \gamma_{i,2,4}r_2r_4 + \gamma_{i,3,3}r_3r_3 + \gamma_{i,3,4}r_3r_4 + \gamma_{i,4,4}r_4r_4 \end{aligned} \tag{3.2}$$

$$\begin{aligned} &\gamma_{i,1,5}r_1x_5 + \gamma_{i,2,5}r_2x_5 + \\ &\gamma_{i,3,5}r_3x_5 + \gamma_{i,4,5}r_4x_5 = x_5(\underbrace{\gamma_{i,1,5}r_1 + \gamma_{i,2,5}r_2 + \gamma_{i,3,5}r_3 + \gamma_{i,4,5}r_4}_{\text{constant } c_{i,5}}) \end{aligned} \tag{3.3}$$

$$\begin{aligned} &\gamma_{i,1,6}r_1x_6 + \gamma_{i,2,6}r_2x_6 + \\ &\gamma_{i,3,6}r_3x_6 + \gamma_{i,4,6}r_4x_6 = x_6(\underbrace{\gamma_{i,1,6}r_1 + \gamma_{i,2,6}r_2 + \gamma_{i,3,6}r_3 + \gamma_{i,4,6}r_4}_{\text{constant } c_{i,6}}) \end{aligned}$$

As the subtraction is the same as the addition in this field, the subtraction is also an xor operation. One optimization therefore is to immediately add all the constant values to the message instead of temporarily saving them. The same applies to the linear terms, here also the values are iteratively added to their proper memory position in the Linear Equation System (LES). After this step the LES is ready for being solved with respect to the o unknowns with $m = o$ equations. This quadratic LES, as in Equation 3.4, can be solved using the Gaussian elimination algorithm.

$$\begin{pmatrix} c_{1,5} & c_{1,6} \\ c_{2,5} & c_{2,6} \end{pmatrix} \cdot \begin{pmatrix} x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} m_1 - v_1 \\ m_2 - v_2 \end{pmatrix} \tag{3.4}$$

3.8.2 Rainbow

Rainbow can be implemented by calculating two differently sized UOV instances. Additionally the transformation T is added which mixes the message parts before the first part of the central map is inverted. This is a simple matrix multiplication of message y by T^{-1}, $y \cdot T^{-1} = x$. The result of the first layer is used to fix the vinegar variables of the second layer. The variables have not to be converted from logarithmic to exponential representation between the layers as they are further progressed in the next layer.

Algorithm 3.8.1: UOV Algorithm

Input : Message y, Message length m, Scheme Parameters v/o
Output : Signature x
begin
`/* Invert the central map */`
Fix first v variables randomly
Solve the LES in o_1 variables with the message as result vector with the Gauss algorithm
`/* Apply` S^{-1} `to the variables */`
Multiply the variables by matrix S^{-1}
Output the variables as the signature

Algorithm 3.8.2: Rainbow Algorithm (Two Layers)

Input : Message y, Message Length m, Scheme Parameters $v/o_1/o_2$
Output : Signature x
begin
`/* Apply` T^{-1} `to message */`
Multiply y by matrix T^{-1}
`/* Invert the central map */`
Fix first v variables randomly
Solve first part of central map in o_1 variables with the first o_1 message-bytes as result vector
Fix $v + o_1$ variables with the solution from the layer above
Solve second part of central map in o_2 variables with the message-bytes o_1 to $o_1 + o_2$ as result vector
`/* Apply` S^{-1} `to variables */`
Multiply the variables by matrix S^{-1}
Output the variables as the signature

3.8.3 enTTS

In case of enTTS, first the Equations 2.5, 2.6 and 2.7 were analyzed and then rewritten in matrix form. The following matrices and equations are the result of this transcription, they hold for the odd enTTS sequence for $l = 5$.

All required values in the matrices in Figure 3.4 and also in Figure 3.6 can be calculated as they depend only on already fixed values. The resulting linear system of equations can be solved by Gaussian elimination. The two middle equations in Figure 3.5 need no equation solving algorithm because they depend exclusively on already known values, the solution for x_{17} and x_{18} can be computed directly. In the third part of enTTS the so-called cross-terms can be subtracted from the message

$$\begin{pmatrix} 1 & p_{8,1}x_1 & p_{8,2}x_2 & p_{8,3}x_3 & p_{8,4}x_4 & p_{8,5}x_5 & p_{8,6}x_6 & p_{8,7}x_7 & 0 \\ 0 & 1 & p_{9,1}x_1 & p_{9,2}x_2 & p_{9,3}x_3 & p_{9,4}x_4 & p_{9,5}x_5 & p_{9,6}x_6 & p_{9,7}x_7 \\ p_{10,7}x_7 & 0 & 1 & p_{10,1}x_1 & p_{10,2}x_2 & p_{10,3}x_3 & p_{10,4}x_4 & p_{10,5}x_5 & p_{10,6}x_6 \\ p_{11,6}x_6 & p_{11,7}x_7 & 0 & 1 & p_{11,1}x_1 & p_{11,2}x_2 & p_{11,3}x_3 & p_{11,4}x_4 & p_{11,5}x_5 \\ p_{12,5}x_5 & p_{12,6}x_6 & p_{12,7}x_7 & 0 & 1 & p_{12,1}x_1 & p_{12,2}x_2 & p_{12,3}x_3 & p_{12,4}x_4 \\ p_{13,4}x_4 & p_{13,5}x_5 & p_{13,6}x_6 & p_{13,7}x_7 & 0 & 1 & p_{13,1}x_1 & p_{13,2}x_2 & p_{13,3}x_3 \\ p_{14,3}x_3 & p_{14,4}x_4 & p_{14,5}x_5 & p_{14,6}x_6 & p_{14,7}x_7 & 0 & 1 & p_{14,1}x_1 & p_{14,2}x_2 \\ p_{15,2}x_2 & p_{15,3}x_3 & p_{15,4}x_4 & p_{15,5}x_5 & p_{15,6}x_6 & p_{15,7}x_7 & 0 & 1 & p_{15,1}x_1 \\ p_{16,1}x_1 & p_{16,2}x_2 & p_{16,3}x_3 & p_{16,4}x_4 & p_{16,5}x_5 & p_{16,6}x_6 & p_{16,7}x_7 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \\ x_{16} \end{pmatrix} = \begin{pmatrix} y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \\ y_{16} \end{pmatrix}$$

Figure 3.4: First part of the enTTS(20,28) central map.

$$\begin{aligned} x_{17} &= y_{17} - p_{17,1}x_1x_6 - p_{17,2}x_2x_5 - p_{17,3}x_3x_4 - p_{17,4}x_9x_{16} - p_{17,5}x_{10}x_{15} - p_{17,6}x_{11}x_{14} - p_{17,7}x_{12}x_{13} \\ x_{18} &= y_{18} - p_{18,1}x_2x_7 - p_{18,2}x_3x_6 - p_{18,3}x_4x_5 - p_{18,4}x_{10}x_{17} - p_{18,5}x_{11}x_{16} - p_{18,6}x_{12}x_{15} - p_{18,7}x_{13}x_{14} \end{aligned}$$

Figure 3.5: Second part of the enTTS(20,28) central map.

$$\begin{pmatrix} 1+p_{19,1}x_0 & p_{19,2}x_{18} & p_{19,3}x_{17} & p_{19,4}x_{16} & p_{19,5}x_{15} & p_{19,6}x_{14} & p_{19,7}x_{13} & p_{19,8}x_{12} & p_{19,9}x_{11} \\ p_{20,1}x_2 & 1+p_{20,2}x_0 & p_{20,3}x_{18} & p_{20,4}x_{17} & p_{20,5}x_{16} & p_{20,6}x_{15} & p_{20,7}x_{14} & p_{20,8}x_{13} & p_{20,9}x_{12} \\ p_{21,1}x_4 & p_{21,2}x_2 & 1+p_{21,3}x_0 & p_{21,4}x_{18} & p_{21,5}x_{17} & p_{21,6}x_{16} & p_{21,7}x_{15} & p_{21,8}x_{14} & p_{21,9}x_{13} \\ p_{22,1}x_6 & p_{22,2}x_4 & p_{22,3}x_2 & 1+p_{22,4}x_0 & p_{22,5}x_{18} & p_{22,6}x_{17} & p_{22,7}x_{16} & p_{22,8}x_{15} & p_{22,9}x_{14} \\ p_{23,1}x_8 & p_{23,2}x_6 & p_{23,3}x_4 & p_{23,4}x_2 & 1+p_{23,5}x_0 & p_{23,6}x_{18} & p_{23,7}x_{17} & p_{23,8}x_{16} & p_{23,9}x_{15} \\ p_{24,1}x_{10} & p_{24,2}x_8 & p_{24,3}x_6 & p_{24,4}x_4 & p_{24,5}x_2 & 1+p_{24,6}x_0 & p_{24,7}x_{18} & p_{24,8}x_{17} & p_{24,9}x_{16} \\ p_{25,1}x_{12} & p_{25,2}x_{10} & p_{25,3}x_8 & p_{25,4}x_6 & p_{25,5}x_4 & p_{25,6}x_2 & 1+p_{25,7}x_0 & p_{25,8}x_{18} & p_{25,9}x_{17} \\ p_{26,1}x_{14} & p_{26,2}x_{12} & p_{26,3}x_{10} & p_{26,4}x_8 & p_{26,5}x_6 & p_{26,6}x_4 & p_{26,7}x_2 & 1+p_{26,8}x_0 & p_{26,9}x_{18} \\ p_{27,1}x_{16} & p_{27,2}x_{14} & p_{27,3}x_{12} & p_{27,4}x_{10} & p_{27,5}x_8 & p_{27,6}x_6 & p_{27,7}x_4 & p_{27,8}x_2 & 1+p_{27,9}x_0 \end{pmatrix} \cdot \begin{pmatrix} x_{19} \\ x_{20} \\ x_{21} \\ x_{22} \\ x_{23} \\ x_{24} \\ x_{25} \\ x_{26} \\ x_{27} \end{pmatrix} = \begin{pmatrix} y_{19} - p_{19,0}x_8x_{10} \\ y_{20} - p_{20,0}x_9x_{11} \\ y_{21} - p_{21,0}x_{10}x_{12} \\ y_{22} - p_{22,0}x_{11}x_{13} \\ y_{23} - p_{23,0}x_{12}x_{14} \\ y_{24} - p_{24,0}x_{13}x_{15} \\ y_{25} - p_{25,0}x_{14}x_{16} \\ y_{26} - p_{26,0}x_{15}x_{17} \\ y_{27} - p_{27,0}x_{16}x_{18} \end{pmatrix}$$

Figure 3.6: Third part of the enTTS(20,28) central map.

as they also depend only on already calculated values. If a system turns out to be not solvable, the whole procedure must be repeated from the first part on with new randomly fixed values.

3.8.4 Gaussian Elimination

The goal of the Gaussian elimination process is to convert the LES into a triangular matrix where the bottom left values are zero. In this triangular form variable after variable from the bottom to the top can be calculated. Valid operations on an LES are for example row exchange, row addition and subtraction and multiplication of a whole row by a constant. With these operations it is possible to form the LES to that specific form.
To form a zero in a column, except at the most upper position, the first value of the row must be equal to all the first values of the rows beneath. Then the rows are subtracted from each other, so that a zero appears at the first position. To achieve a state where the values in the first column all equal the value in the most upper position, every row must be multiplied by an individual factor. The factor can be calculated by division of the most upper value in the current column by the first value in the row. By repeating this step for every row, zeros are generated in the first column beneath the first value. If the same is repeated with the second value

Algorithm 3.8.3: enTTS Algorithm

Input : Message y, Scheme Parameter l
Output : Signature x
begin
  /* Apply T^{-1} to message */
  Multiply y by matrix T^{-1}
  /* Invert the central map */
  for $i = 1$ *to* $2l - 3$ **do**
    fix x_i randomly
  Solve first LES to get x_{2l-2} to x_{4l-4}
  if *not solvable* **then**
    begin from start
  Solve the two middle polynomials to obtain x_{4l-3} and x_{4l-2}
  Fix x_0 randomly
  Solve second LES to get x_{4l-1} to x_{6l-3}
  if *not solvable* **then**
    begin from start
  /* Apply S^{-1} to variables */
  Multiply the variables by matrix S^{-1}
  Output the variables as the signature

of the second column and so on, the system is formed into a triangular shape.
To optimize the required memory the LES is brought into Lower Upper (LU) form. For every division a place to store the coefficient of the following row multiplication is required. Instead of writing this value into a new memory space, the value can be saved at the position where the zero would occur. When the lower left triangle with the coefficients and the upper right triangle with the remaining values are separated, the so called LU form is complete. The system in the LU form is ready to be solved. First the coefficients are multiplied by values of the solution vector, which is called forward substitution. Then, beginning with the last equation which now contains only one unknown variable, it can easily be solved and the result can be substituted into all other equations; this step is called backward substitution. With this procedure a solution for all unknowns can be calculated.
Since the output of the division could be zero, a multiplication of a row by zero would cause a "zero-row" and a faulty result. Therefore, this special case must be caught. If a zero value occurs an option would be to assign new values to the vinegar variables. Another option would be pivoting, find another element which produces no zero and exchange the columns. This would cost additional computation time, and a reversion of the swap would also be necessary. For simplification the first attempt is chosen.
The solution is written directly into the result vector, the right side of the LES, in-

stead of a new memory space. This overwriting is admissible because the right-hand side contained the message, which was saved to another location before this step.

3.8.5 RAM Requirements

$\mathcal{MQ}$ schemes do not need a lot of RAM, in contrast to their persistent flash memory requirements. In Table 3.2 the requirements are listed. Besides the small amount of RAM required for single persistent, counting or temporary variables, only the Gaussian elimination algorithm needs a noticeable amount of RAM. As the inversion is computed in place, only one quadratic system at a time must be stored in RAM. In case of multiple layers the maximal requirements are defined by the largest system of equations.

Table 3.2: Minimal Ram Requirements for LES Solving in Bytes

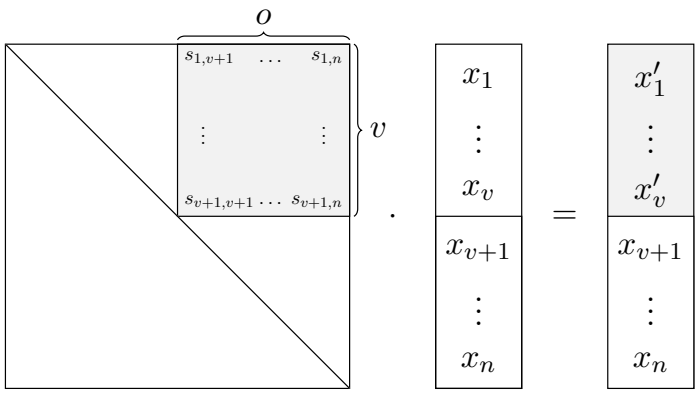
security	2^{64}	2^{80}	2^{128}	general
UOV	441	784	1936	m^2
Rainbow	400	729	1849	$(o_1 + o_2)^2$
enTTS	169	289	841	$(2l - 1)^2$

3.9 Inverting the Linear Transformations

To invert a linear transformation its coefficient matrix must be inverted. This can be accomplished on the PC by Gaussian elimination. The matrix is brought only once into LU form and then solved for each column of the identity matrix as the solution vector. A random $n \times n$ matrix is solvable with high probability. The inverted matrices S and T are then saved on the μC as parts of the private key together with the central map on the μC. During signature generation only the multiplication with the inverse matrices must be computed.
If the matrix S was chosen self-invertible, the $n \times n$ multiplication can be reduced to n copy operations and a multiplication of a $v \times o$ matrix by a vector of o variables as shown in Figure 3.7. Note that only the vinegar and not the oil variables are altered by S.

3.10 Key Generation

In Section 2.6 the key generation was explained theoretically. In this section the implementation details are shown. As mentioned before there are two different ways of generating a key pair. In this work the approaches are called “forward” and



$$x'_i = \begin{cases} x_i + x_{v+1}s_{i,v+1} + x_{v+2}s_{i,v+2} + \ ... + x_n s_{i,n} & , 0 \leq i \leq v \\ x_i & , v+1 \leq i \leq n \end{cases}$$

Figure 3.7: This figure illustrates the reduction of multiplications in signature generation when a self invertible S is used.

"backward" because of the order in which $\mathcal{F}$ and $\mathcal{P}$ are generated.

3.10.1 Forward

The "forward" approach defines $\mathcal{F}, T$ and S and then calculates the composition of these three matrices to obtain $\mathcal{P}$ as described in Equation 2.8. This approach can be applied to all schemes regardless of the form of the central equations and existence of the second linear transformation T. Matrix S is chosen self-invertible in the case of UOV and Rainbow, as described in Section 3.6.3.

3.10.2 Backward

The other "backward" approach defines S and a large part of $\mathcal{P}$ named B, then computes $\mathcal{F}$ and calculates the remaining part of $\mathcal{P}$. This approach uses the special structure in UOV. It would also be possible to apply this technique to Rainbow with a kind of layered B, due to the transformation T some benefits could not be used, but this is not in the scope of this work. It is further possible to choose B fixed from another field as this part is not security relevant in terms of key recovery attacks according to [PTBW11]. S can be chosen self-invertible as in the forward approach.

Algorithm 3.10.1: Forward Key Generation

Input : Message Length m, Scheme Parameters o/v , $v/o_1/o_2$ or l
Output : Private and Public Maps
begin

```
/* Generate random linear maps                                    */
```
for *Elements in* S **do**
  fill with random values

Invert S; **if** S *is not invertible* **then**
  Choose S again

for *Elements in* T **do**
  fill with random values

Invert T; **if** T *is not invertible* **then**
  Choose T again

```
/* Generate Random Central Map                                    */
```
for *Elements in* $\mathcal{F}$ **do**
  Fill with random values

```
/* Combine the linear map with the central map                    */
```
for *Each polynomial in* $\mathcal{F}$ **do**
  Calculate the composition with S

Multiply all polynomials in $\mathcal{F}$ by T to obtain $\mathcal{P}$

Save $S^{-1}, \mathcal{F}, T^{-1}$ as the private key
Save the composition $\mathcal{P}$ as the public key

The following toy example for the 0/1 UOV with $(o, v) = 3, 6$ should give a better understanding of the whole procedure. The number of variables is $n = o + v = 9$, the number of oil variables equals the message length $o = m$, the dimension of A is $D' \times D'$ with $D' = D + D_2 = 45$ where $D = \frac{v \cdot (v+1)}{2} + o \cdot v = 39$ and $D_2 = \frac{o \cdot (o+1)}{2} = 6$ The key generation starts with the linear map S which is chosen randomly from $\mathbb{F}_{2^8}$ and is additionally chosen self-invertible. The matrix B is randomly generated as stated in Section 2.6. In Figure 3.8 the structures of both are readily identifiable. Matrix A is generated according to Equation 2.9 and a permutation is applied to the rows according to a suitable complementary Turań graph. To be able to multiply matrix A by $\mathcal{F}$, the matrix must be trimmed to a $D \times D$ matrix after permutation, as visualized in Figure 3.9. The trimmed matrix A' can now be inverted. How this inversion changes the structure can be seen in Figure 3.10. The result of the inversion is verified by performing a test multiplication of A by A^{-1} and a check if the result equals the identity matrix. Having all the necessary matrices, $\mathcal{F}$ and

Algorithm 3.10.2: Backward Key Generation

Input : Message Length m, Scheme Parameters o/v, Field $\mathbb{F}$
Output : Private and Public Maps
begin
  `/* Generate random linear maps */`
  for *Elements in* S **do**
    fill with random values
  Invert S; **if** S *is not invertible* **then**
    Choose S again

  if *Field is* $\mathbb{F}_2$ **then**
    Generate complementary Turań graph CT
    Permute A according to the CT
    Choose B randomly from $\mathbb{F}_2$
  else
    Choose B randomly from $\mathbb{F}_{2^8}$
  Generate A from S
  Invert and trim A

  `/* Calculate Central Map */`
  Calculate $\mathcal{F} = A^{-1} \cdot B$

  `/* Calculate Public Map */`
  for *Each polynomial in* $\mathcal{F}$ **do**
    Calculate the composition with S

  Save $S^{-1}, \mathcal{F}$ as the private key
  Save the composition $\mathcal{P}$ as the public key

$\mathcal{P}$ can now be computed as shown in Figure 3.11. Here it is apparent that the $v \times v$ parts of $\mathcal{F}$ and $\mathcal{P}$ are equal. It also becomes obvious that most polynomials in $\mathcal{P}$ are from $\mathbb{F}_2$ and some from $\mathbb{F}_{2^8}$. The positions correspond to the edges in the complementary Turań graph and do not have to be kept secret.

3.11 Generic Code

As mentioned before in Section 3.3.2 it is important to be able to increase parameters to match new or faster attacks. Therefore, the code in this work should be easily parametrizable. To this end, the code uses constant definitions. At the begin of the code all important sizes and values are recursively defined by `C #define`

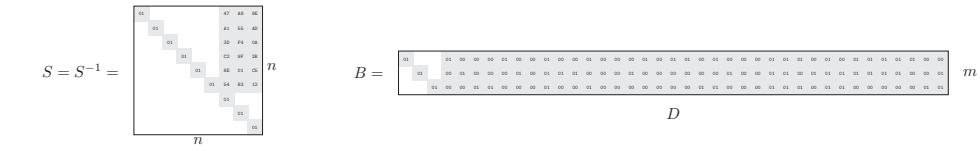


Figure 3.8: Step 1: Generate S and B

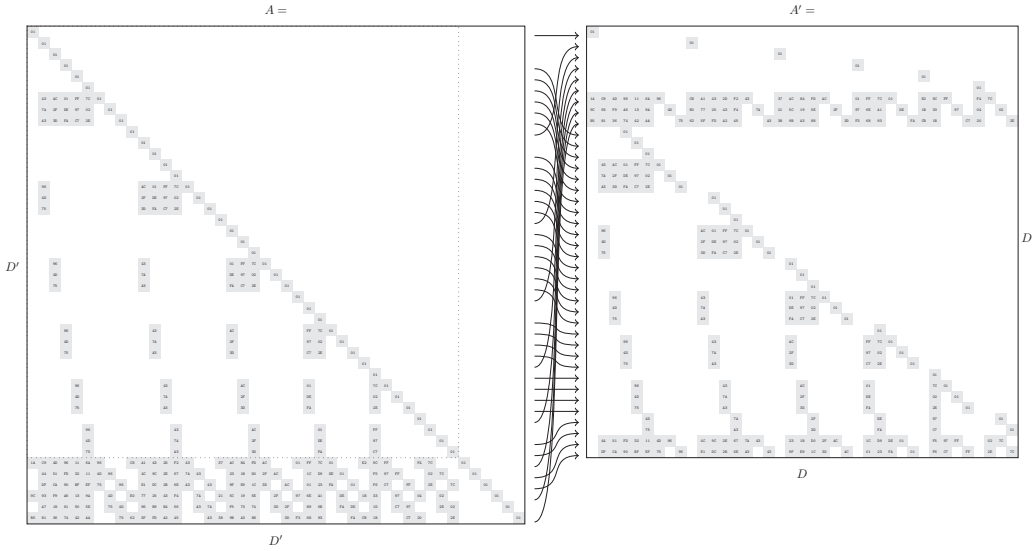


Figure 3.9: Step 2: Generate A and permute rows.

declarations. All loops use these definitions.
Because the enTTS implementation in this work is based on unrolled loops, the code is written not as generic as in the case of UOV and Rainbow. To be able to generate new enTTS instances, a generic code generator was implemented. The cyclic structures in enTTS seemed inefficient for implementation as loops. Additionally, loops for the small amount of equations would be a significant overhead. For very large enTTS instances the unrolled versions produce code that is quite long. In Table 4.1 and Table 4.2 the achieved code sizes are summarised.

3.12 Problems

To read the keys out from internal flash memory the build-in avr libc function `pgm_read_byte_far()` is used. This function expects a 32 bit long address and returns an 8 bit value. One problem arises from the avr-gcc compiler: if a simple conversion from a symbol to an `uint32` takes place, the upper half of the address

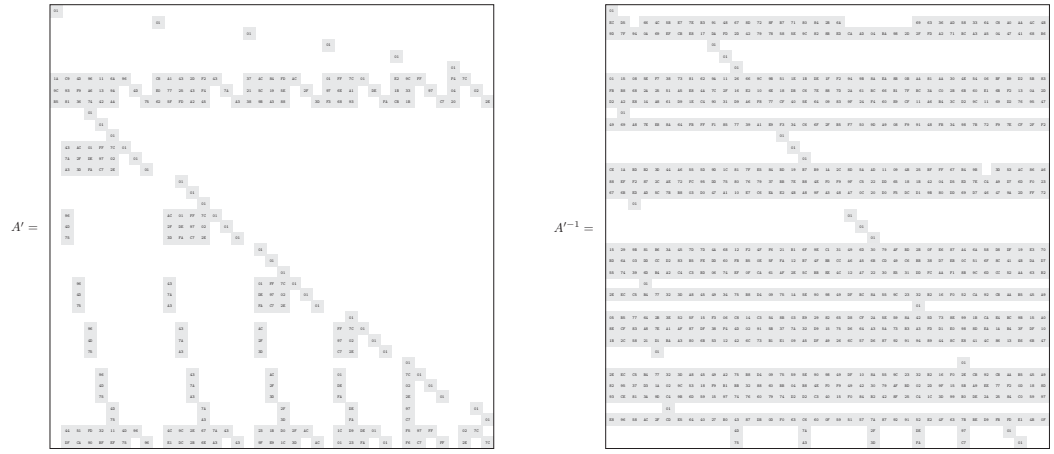


Figure 3.10: Step 3: Invert A.

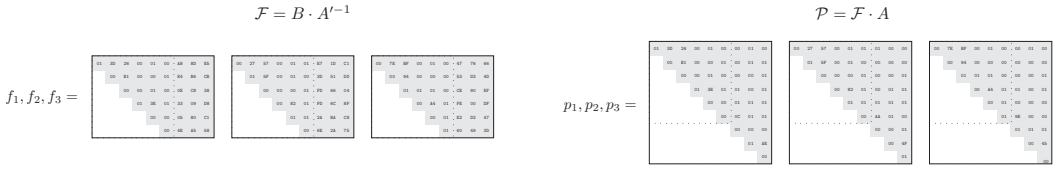


Figure 3.11: Step 4: Compute $\mathcal{F}$ and $\mathcal{P}$.

is lost because of the 32 bit address incompatibility of the avr-gcc. The solution to this is a macro which converts a symbol to `uint32`. The following code is used to load the appropriate values into the proper address registers. Bits 24 to 31 are not needed and therefore left out.
If a key must be read out, the pointer must be grabbed with the macro in Listing 3.1 and must then be saved to a 32 bit variable. By increasing this variable iteratively every byte of the key can be reached.

```
// Macro to access data defined in PROGMEM above 64kB
#define GET_FAR_ADDRESS(var)                    \
({  uint32_t tmp;                               \
    __asm__ __volatile__(                       \
            "ldi %A0, lo8(%1)"          "\n\t"  \
            "ldi %B0, hi8(%1)"          "\n\t"  \
            "ldi %C0, hlo8(%1)"         "\n\t"  \
        :  "=d" (tmp)                           \
        :  "p"  (&(var))                        \
    );                                          \
    tmp; })                                     \
```

Listing 3.1: Macro For Far Addresses

To store such a large array of bytes in the memory, it is necessary to store the key into a GNU assembler code file as a workaround. The approach to save this key into a C-array fails because of the WinAVR 16 bit addressing boundary, which does not admit arrays with elements that cannot be addressed by a 16 bit address. The assembly file contains all the bytes in a row so that the linker accepts it as a block of values and no addressing issues occur.
In such a file the starting point of every key is declared by a global symbol followed by the key bytes. If the keys start at an odd address, an error occurs at compile time. To fix this `.align 2` must be used. This instructs the linker to align the keys at an even address. By inserting the keyword `.text` the values are not copied to the SRAM and remain in the flash memory. The complete required file format is shown in Listing 3.2.

```
.align 2
.text
.global private_key_f
private_key_f:
.byte 0xDC, 0xBB, 0x2D, 0xD4, 0x55, 0x23, 0xB5, 0x3A, ...
.byte 0x0C, 0x3C, 0x73, 0xA7, 0xF0, 0x18, 0x85, 0xC4, ...
.byte  ...
```

Listing 3.2: Start of `privateKey.S`

4 Results

Table 4.1 and Table 4.2 show the achieved results. They are easy to compare because schemes are grouped by security level. For all schemes key size, runtime and code size are given. Where applicable the system parameter size is also included. The public and secret key sizes can be easily calculated. One element responds to one byte and no other overhead needs to be saved so the keys consist only of the coefficients of the public or secret maps and the linear transformations. In the case of 0/1 UOV a large part is fixed and declared as a system parameter, but it must be saved anyway or be easy to generate in a real-world scenario, therefore this size is also listed.
Clock cycles are counted internally with two concatenated 16 bit counters which are enabled to count on every clock cycle. If a system appeared not to be solvable the clock was restarted and the random variables were chosen again. The values in the tables might be tuned significantly more by using another compiler or different compile options. Further, in all schemes the design decision is made for speed not for code size, so there is still room for improvements.

4.1 Sign

When comparing signature generation characteristics, the probably most important ones are signature generation time and secret key size. The mostly small code size is negligible on a μC, only the unrolled enTTS instances do not scale well. Comparing private key sizes, enTTS performs the best with its very small private keys. The private key size adds up to only about 30% of the key size for rainbow with 2^{64} bit security. With raising parameters the key size scales even better, it is for 2^{128} bit security only 13% of the rainbow key size. Also in terms of time for signature generation enTTS is better, in the 2^{80} set, where enTTS compared to Rainbow signs about 2.8 times faster and compared to 0/1 UOV 5.8 times faster. Between UOV and 0/1 UOV, which has a private key with elements also from $\mathbb{F}_2$, the difference is only 1.2 ms. Even taking the system parameter not into account, Rainbow performs in all categories better than 0/1 UOV except with respect to the code size. The code is about twice as long because in Rainbow the Gaussian elimination was implemented twice, for each layer separately, to improve the signature time. If the code size is important it could be reduced by half by using a Gaussian elimination algorithm with variable dimensions. The message to signature ratio is another factor which is often compared in signature schemes. UOV has the highest expansion factor, with a message to signature ratio of approximately 2.3, followed by Rainbow with 1.7, and

enTTS with 1.4. When measuring scalability for secret key size at the step from 2^{64} to 2^{128}, UOV and 0/1 UOV have an increase factor of 9, Rainbow 10 and enTTS 4.

Table 4.1: Results for signature generation

	Scheme	n	m	private Key [Byte]	Parameter [Byte]	Clockcyles x 1000	Time[ms] @32MHz	Code Size [Byte]
	enTTS$(5, 20, 28)$	28	20	1351	*	153	4.79	12890
	enTTS$(5, 20, 28)$[YCCC06]	28	20	1417	*	568[1]	17.75[2]	-
2^{64}	UOV$(21, 28)$	49	21	21462	*	1,615	50.49	2188
	0/1 UOV$(21, 28)$	49	21	12936	8526	1,577	49.29	2258
	Rainbow$(15, 10, 10)$	35	20	9250	*	848	26.51	4162
	enTTS$(7, 28, 40)$	40	28	2731	*	332	10.37	24898
2^{80}	UOV$(28, 37)$	65	28	49728	*	3,637	113.66	2188
	0/1 UOV$(28, 37)$	65	28	30044	19684	3,526	110.20	2258
	Rainbow$(18, 13, 14)$	45	27	19682	*	1,740	54.38	4162
	enTTS$(9, 36, 52)$	52	36	4591	*	609	19.03	41232
2^{128}	UOV$(44, 59)$	103	44	194700	*	13,314	416.07	2188
	0/1 UOV$(44, 59)$	103	44	116820	77880	12,782	399.43	2258
	Rainbow$(36, 21, 22)$	79	43	97675	*	8,227	257.11	4162
	enTTS$(15, 60, 88)$	88	60	13051	*	2,142	66.94	116698

* Not applicable

[1] Derived from values in original work

[2] Scaled to the same clock frequency

4.2 Verify

The verification algorithm is very similar in each scheme, only 0/1 UOV has another algorithm for $\mathbb{F}_2$ elements. Therefore, the key size and time only depends on the parameters. As UOV and Rainbow use the exact same code for verification their sizes are equal, 0/1 UOV has a slightly longer code due to the additional check for $\mathbb{F}_2$ elements. In enTTS the linear terms in $\mathcal{P}$ were not removed, the increase from the additional verification code is visible. The smallest public key without taking generated or cyclic keys into account is Rainbow's with a key of about 56% the size of enTTS and 47% the size of UOV. Without the system parameter part, 0/1 UOV has the smallest public key by far. Because of the relatively high values of the parameters n and m enTTS scales bad in the size of the public key. The verification time rankings can be almost directly applied to the size of the public key. Only 0/1 UOV and Rainbow switch places because the system parameter must also be verified, but there is certainly room for embedding a much better structure for implementations in $\mathcal{P}$, which could give the advantage back to 0/1 UOV. None the less the small parameters in Rainbow show an effect on the verification times. The gain of verification time of 0/1 UOV in comparison to the standard UOV is only minimal as the multiplication by table look-up method has no significant runtime

difference in comparison to a multiplication with 0 or 1 as the 0 case is a special case and is checked anyway every time in a normal multiplication in F_{2^8}.
Again the scalability for doubling the security parameter can be listed, which is for UOV and 0/1 UOV a factor of 9, for Rainbow 11 and for enTTS 10.
When measuring scalability for secret key size at the step from 2^{64} to 2^{128}, UOV and 0/1 UOV have an increase factor of 9, Rainbow 10 and enTTS 4. UOV scales the best with respect to the public key size.

Table 4.2: Results for verification

	Scheme	n	m	public Key [Byte]	Parameter [Byte]	Clockcyles x 1000	Time[ms] @32MHz	Code Size [Byte]
	enTTS$(5, 20, 28)$	28	20	8120	*	1,126	35.22	827
	enTTS$(5, 20, 28)$[YCCC06]	28	20	8680	*	5,808[1]	181.5[2]	-
2^{64}	UOV$(21, 28)$	49	21	25725	*	1,690	52.83	466
	0/1 UOV$(21, 28)$	49	21	4851	20874	1,395	43.60	578
	Rainbow$(15, 10, 10)$	35	20	12600	*	1,010	31.58	466
	enTTS$(7, 28, 40)$	40	28	22960	*	2,558	79.95	827
2^{80}	UOV$(28, 37)$	65	28	60060	*	3,911	122.23	466
	0/1 UOV$(28, 37)$	65	28	11368	48692	3,211	100.37	578
	Rainbow$(18, 13, 14)$	45	27	27945	*	2,214	69.19	578
	enTTS$(9, 36, 52)$	52	36	49608	*	6,658	208.07	827
2^{128}	UOV$(44, 59)$	103	44	235664	*	14,134	441.70	466
	0/1 UOV$(44, 59)$	103	44	43560	192104	13,569	424.04	578
	Rainbow$(36, 21, 22)$	79	43	135880	*	9,216	288.01	466
	enTTS$(15, 60, 88)$	88	60	234960	*	3,0789	962.17	827

* Not applicable
[1] Derived from values in original work
[2] Scaled to the same clock frequency

4.3 Comparison With Other Schemes

A comparison of a μC with an ASIC or PC implementation is hard, because on these platforms instructions can be executed in parallel and the instruction set is much richer. The only $\mathcal{MQ}$ implementation this implementation can be compared to is the one from [YCCC06]. The authors implemented enTTS$(5, 20, 28)$ on a MSP430 running at 8 MHz. Their signature generation requires 17.75 ms and verification 181.5 ms when scaled up to our clock frequency. Although, the MSP430 is a 16 bit CPU, our implementation is a factor of 3.7 faster in signing and 5.1 times faster in verifying.
Compared to the implementation of the classical signature schemes RSA and ECDSA, the implementations of this work perform well. For the security level of 2^{80} bit [GPW+04] reports 203ms for an ECC sign operation, where enTTS is two to ten times faster. For the verifying operation our work is up to three times faster. Due

to the short exponent in RSA-verify, [GPW+04] verifies in the same order of magnitude. But the RSA-sign operation is at least a factor of 25 slower than our work. Table 4.3 summarizes other implementations on comparable 8 bit platforms.

Table 4.3: Overview of other implementations on comparable platforms.

Method	Time[ms]@32MHz	
	sign	verify
enTTS$(5, 20, 28)$[YCCC06]	17.75[1]	181.5[1]
ECC-P160 (SECG) [GPW+04]	203[1]	203[1]
ECC-P192 (SECG) [GPW+04]	310[1]	310[1]
ECC-P224 (SECG) [GPW+04]	548[1]	548[1]
RSA-1024 [GPW+04]	2,748[1]	108[1]
RSA-2048 [GPW+04]	20,815[1]	485[1]
NTRU-251-127-31 sign [DPP08]	143[1]	-

[1] For a fair comparison with our implementation running at 32MHz, timings at lower frequencies were scaled accordingly.

5 Conclusion

The results show that $\mathcal{MQ}$ schemes can compete or even be better than current standard algorithms for digital signatures. Every scheme implemented in this work has its own characteristics with advantages and disadvantages. The enTTS scheme is currently the fastest signature generating scheme with its very small private polynomials. In terms of verification speed Rainbow is the best choice, its layered structure allows small parameters which lead to a small public key and therefore to a fast verification.
These results give for the first time a realistic and fair estimation of the performance of UOV, 0/1 UOV, Rainbow, and enTTS for two main reasons. On the one hand, every scheme was benchmarked with equal security levels and parameters, which are based on the newest known attacks; on the other hand, they were implemented on the same platform and using the same coding style including the same arithmetics and optimizations.
The steps that were necessary to carry out this comprehensive comparison are explained in the following summary.

5.1 Summary

To be able to design such schemes they first had to be analyzed theoretically. Important components like linear map inversion, central map inversion, Gaussian elimination, and others were identified and assembled to algorithms. One important task was to analyze the central maps which are different in every scheme. The implementation was designed in a way that allows for the simple modification of the central maps which renders the implementation of new schemes a lot easier and faster.
After the theory and design decisions were outlined, the implementation of the components was explained. Important details and differences to the other schemes were pointed out. Were possible, a visualization was given to assist the understanding. With a focus on implementation, mathematical details were explained only to a level in which they were necessary for a good implementation, for further information references were given to the original papers.
At the end a comprehensive comparison of four schemes in three security levels in at least three characteristics was performed and evaluated. Based on the results further development can be inspired, some ideas follow in the next chapter.

5.2 Further Improvements / Future Work

There is still space for improvements and the end is not reached yet. A few ideas were not implemented in this work. Saving the system parameters is not optimal as mentioned several times before. Here, a replacement by a pseudo-random number generator or another generator function or cyclic key would reduce the public key even more. In this 0/1 UOV implementation all elements of $\mathbb{F}_2$ are saved as a one byte value. It would be possible to achieve smaller keys when saving eight elements in one byte. Combined with a verification function which utilizes assembler instructions maybe even a faster verification could be possible.
An overall time versus code size trade-off is still a topic to investigate, $\mathcal{MQ}$ schemes are very well scalable with regard to this trade-off.
The development of new schemes is not at the end, a scheme which combines all advantages of enTTS and Rainbow or balances them to a middle scheme is probably possible, in particular because the research on new attacks is making fast progress. With progressing quantum computer technology it remains to be seen how, when, and if $\mathcal{MQ}$ schemes become an alternative to new standards. This work aims to provide a step in the right direction.

A Appendix

```
uint16_t get_seed()
{
  uint16_t seed = 0;  //initialise seed with 0
  uint16_t *p = (uint16_t*) (RAMEND+1); //Pointer SRAM end
  extern uint16_t __heap_start; //Pointer SRAM start

  while (p >= &__heap_start + 1) //while p doesn't reach HEAP
    seed ^= * (--p);        //xor the value on seed, decrement p

  return seed;
}
```

Listing A.1: Seed generation from uninitialised SRAM

Acronyms

UOV Unbalanced Oil and Vinegar

$\mathcal{MQ}$ Multivariate Quadratic

LES Linear Equation System

LU Lower Upper

μ**C** micro controller

IP Isomorphism of Polynomials

PKS Public Key System

List of Figures

List of Tables

Bibliography

[20004] Tts: High-speed signatures on a low-cost smart card. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, pages 371–385, 2004.

[Atma] Atmel. At90usb1287 website. `http://www.atmel.com/dyn/products/product_card_v2.asp?part_id=3875`.

[Atmb] Atmel. Atxmega128 website. `http://www.atmel.com/dyn/products/product_card.asp?part_id=4298`.

[Atmc] Atmel. Xplain website. `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4506`.

[BCB+08] S. Balasubramanian, H.W. Carter, A. Bogdanov, A. Rupp, and Jintai Ding. Fast multivariate signature generation in hardware: The case of rainbow. In *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, pages 25 –30, july 2008.

[BD08] Johannes Buchmann and Jintai Ding, editors. *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, volume 5299 of *Lecture Notes in Computer Science*. Springer, 2008.

[BFMR11] Charles Bouillaguet, Pierre-Alain Fouque, and Gilles Macario-Rat. Practical key-recovery for all possible parameters of sflash. In *ASIACRYPT*, pages 667–685, 2011.

[BFP09] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Hybrid approach for solving multivariate systems over finite fields. *Journal of Mathematical Cryptology*, volume 3(issue 3):177–197, 2009.

[CD03] Nicolas T. Courtois and Magnus Daum. On the security of hfe, hfev- and quartz. In *In Proceedings of PKC 2003, volume 2567 of LNCS*, pages 337–350. SpringerVerlag, 2003.

[cFJ03] Jean charles Faugère and Antoine Joux. Algebraic cryptanalysis of hidden field equation (hfe) cryptosystems using gröbner bases. In *In Advances in Cryptology, CRYPTO 2003*, pages 44–60. Springer, 2003.

[DPP08] Benedikt Driessen, Axel Poschmann, and Christof Paar. Comparison of Innovative Signature Algorithms for WSNs. In *Proceedings of ACM WiSec 2008*, ACM, 2008.

[DS05] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *ACNS*, pages 164–175, 2005.

[GC00] Louis Goubin and Nicolas Courtois. Cryptanalysis of the ttm cryptosystem. In *Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '00, pages 44–57, 2000.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability — A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979. ISBN 0-7167-1044-7 or 0-7167-1045-5.

[GPW+04] Nils Gura, Arun Patel, Arvinderpal W, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. pages 119–132, 2004.

[KPG99] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced Oil and Vinegar signature schemes. In *Eurocrypt*, pages 206–222, 1999.

[KS98] Aviad Kipnis and Adi Shamir. Cryptanalysis of the oil and vinegar signature scheme. In *Proceedings of CRYPTO'98, Springer, LNCS n o 1462*, pages 257–266. Springer Verlag, 1998.

[KS99] Aviad Kipnis and Adi Shamir. Cryptanalysis of the hfe public key cryptosystem by relinearization. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 19–30, 1999.

[Pat95] Jacques Patarin. Cryptoanalysis of the matsumoto and imai public key scheme of eurocrypt'88. In *CRYPTO*, pages 248–261, 1995.

[PBB10a] Albrecht Petzoldt, Stanislav Bulygin, and Johannes Buchmann. A multivariate signature scheme with a partially cyclic public key. In Carlos Cid and Jean-Charles Faugere, editors, *Proceedings of the 2nd International Conference on Symbolic Computation and Cryptography (SCC 2010)*, pages 229–235, Jun 2010.

[PBB10b] Albrecht Petzoldt, Stanislav Bulygin, and Johannes Buchmann. Selecting parameters for the Rainbow signature scheme. In *PQCrypto*, pages 218–240, 2010.

[Pro] LUFA Project. Lufa project website. `http://fourwalledcubicle.com/LUFA.php`.

[PTBW11] Albrecht Petzoldt, Enrico Thomae, Stanislav Bulygin, and Christopher Wolf. Small public keys and fast verification for multivariate quadratic public key systems. In *CHES*, pages 475–490, 2011.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. on Computing*, pages 1484–1509, 1997.

[TW12] Enrico Thomae and Christopher Wolf. Solving underdetermined systems of multivariate quadratic equations revisited. In *Practice and Theory in Public Key Cryptography (PKC 2012)*. Springer-Verlag, 2012.

[TYD+11] Shaohua Tang, Haibo Yi, Jintai Ding, Huan Chen, and Guomin Chen. High-speed hardware implementation of rainbow signature on fpgas. In Bo-Yin Yang, editor, *PQCrypto*, volume 7071 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2011.

[WBP04] Christopher Wolf, An Braeken, and Bart Preneel. Efficient cryptanalysis of rse(2)pkc and rsse(2)pkc, 2004.

[YC04] Bo-Yin Yang and Jiun-Ming Chen. Tts: Rank attacks in tame-like multivariate pkcs. Cryptology ePrint Archive, Report 2004/061, 2004. `http://eprint.iacr.org/`.

[YCCC06] Bo-Yin Yang, Chen-Mou Cheng, Bor-Rong Chen, and Jiun-Ming Chen. Implementing minimized multivariate pkc on low-resource embedded systems. In John Clark, Richard Paige, Fiona Polack, and Phillip Brooke, editors, *Security in Pervasive Computing*, volume 3934 of *Lecture Notes in Computer Science*, pages 73–88. Springer Berlin / Heidelberg, 2006. 10.1007/11734666.