Efficient Implementations of Multivariate Quadratic Systems[⋆]

Côme Berbain, Olivier Billet, and Henri Gilbert

France Télécom R&D
38–40, rue du Général Leclerc
92794 Issy les Moulineaux Cedex 9 — France
`forname.lastname@orange-ftgroup.com`

Abstract. This work investigates several methods to achieve efficient software implementations of systems of multivariate quadratic equations. Such systems of equations appear in several multivariate cryptosystems such as the signature schemes SFLASH, Rainbow, the encryption scheme PMI$^+$, or the stream cipher QUAD. We describe various implementation strategies. These strategies were combined to implement the public computations of three asymmetric schemes as well as the stream cipher QUAD. We conducted extensive benchmarks on our implementations which are exposed in the final section of this paper. The obtained figures support the claim that when some care is taken, multivariate schemes can be efficiently implemented in software.

Keywords: multivariate systems, quadratic equations, efficient software implementation.

1 Introduction

Multivariate cryptography is a quickly expanding research branch of cryptology. Its development, initiated by the seminal work of T. Matsutomo and H. Imai [7,10,11] and J. Patarin [13,14,15], was mainly motivated by the search for alternatives to arithmetic asymmetric cryptosystems such as RSA. Multivariate cryptography exploits the intractability of solving a multivariate system of low degree equations (typically quadratic equations) over a small finite field. Many multivariate asymmetric schemes for encryption, signature, or authentication have been proposed over the past years and a restricted number of them (e.g. the SFLASH and UOV signature schemes [1,8]) have successfully resisted cryptanalysis so far.

The development of multivariate cryptography has recently taken another path with the proposal of a symmetric multivariate algorithm, the stream cipher

[⋆] The work described in this paper has been supported by the French Ministry of Research RNRT X-CRYPT project and by the European Commission through the IST Program under Contract IST-2002-507932 ECRYPT.

QUAD [2]. This cipher takes advantage of the specific characteristics of multivariate systems of equations in order to provide some provable security properties (an extremely unusual feature in the context of symmetric cryptography) at the expense of a moderate performance penalty.

What renders multivariate cryptography attractive from an implementation point of view is that intractable multivariate problems can be extremely compact. Consequently, the performance of multivariate schemes' implementations is often intermediate between the typical performance of asymmetric schemes and the typical performance of symmetric schemes. But efficient implementations of multivariate schemes have not been systematically investigated so far, and there is a lack of figures to serve as reference when comparing multivariate schemes to other systems in terms of practicality. In the case of the asymmetric multivariate signature scheme SFLASH for instance, considerable optimization efforts [1] were made to produce a highly efficient implementation of the secret key computations and to establish that unlike RSA, the SFLASH signature algorithm can be efficiently embedded in 8-bit smart cards without coprocessor. However there was no effort to optimize the public key computations which are typically done on a server. Another example where optimization takes place on the secret's holder side and does not relate to any public computations is the study of public key generation presented in [17].

The main issue one is faced when implementing multivariate schemes in software is to achieve an efficient computation of (at least apparently) random systems of quadratic equations over a small finite field GF(q). This issue arises for instance in software implementations of the public computations in the setting of asymmetric schemes (for signature verification, encryption, or entity authentication purposes), or for the implementation of symmetric schemes (like in the case of the stream cipher QUAD). Efficient implementations of such multivariate quadratic systems of equations without using any specific structure will thus benefit all multivariate schemes as it is not tight to any particular cryptosystem.

This paper is organized as follows. Section 2 describes several methods that can be used to efficiently implement such generic systems. We discuss the special cases of GF(2), GF(2^4), and GF(2^8) which are in practice the most suitable ground fields in most multivariate cryptosystems, and focus on parameter sizes (like the number of unknowns and the number of polynomials in the system) directly arising from real cryptosystems outlined in Section 3. Section 4 exhibits a comprehensive set of benchmarks showing the performance of our various C ANSI implementations of asymmetric schemes like SFLASH, PMI$^+$, and Rainbow public keys as well as QUAD's internal system of equations. We finally draw our conclusions.

2 Implementation Strategies

This section describes various strategies we investigated in order to efficiently implement the computation of quadratic systems in several cases of cryptographic significance. We specify along these descriptions which strategy seems best suited for a particular setting. The computation of any multivariate system of m quadratic equations in n unknowns over a finite field $\mathrm{GF}(2^p)$ can obviously be split into two phases: generating the value of each of the degree two monomials, and actually computing the value of the output polynomials. While for both steps there are rather academic ways to perform the computations, they have to be tuned to the context of use. All the algorithms described hereafter have the same asymptotic complexity, namely $O(n^2)$ to generate the monomials and $O(mn^2)$ to compute the polynomials. However because we target real cryptographic schemes, the values of m and n lie in some range and as we show in the following it is possible to achieve big speedups. This is especially true when, as is the case in our practical cryptographic examples, the values of n and m are the same order of magnitude as the machine word size w. Moreover, fine tuning the implementation in order to take into account the available amount of L2 cache plays an essential role in the overall efficiency.

2.1 Generating All Monomials

There are many ways to compute the values of all degree two monomials for a given set of n variables over a finite field $\mathrm{GF}(2^p)$. However, their respective efficiency highly depends on the ground field size. We hereafter focus on the natural cases $p \in \{1, 4, 8\}$.

The Naïve Way. The most naïve way to compute a set of monomials is obviously to consider every pair (x_i, x_j) of variables in turn and to generate the corresponding monomial $x_i x_j$. This method is efficient provided one has direct access to each variable, which is the case for instance when working over $\mathrm{GF}(2^8)$. On the opposite, when binary variables are packed in big words, the overhead of accessing the variables is prohibitive.

Rotations. Another intuitive way which may seem particularly attractive in the case of a binary ground field (since the total number of operations is reduced by a factor of the machine word size) is to consider the set of variables as a vector of machine words and to perform w multiplications in parallel using bitwise `ands` between cyclically rotated versions of this vector. However this is not the most efficient strategy over $\mathrm{GF}(2)$ as is shown in the sequel.

Bitslice Multiplications. This strategy can be rather efficient in the case of intermediate ground fields such as $\mathrm{GF}(2^4)$ provided we implement a bitslice multiplication to replace the bitwise multiplications of the binary case. Such a bitslice implementation is described for instance in [9]. It basically requires storing the set of n variables over $\mathrm{GF}(2^4)$ in p vectors of size $\lceil n/w \rceil$ words and performing the computations on these vectors directly, hence avoiding bit level manipulations. (In the existing cryptographic schemes, the number n of variables is about the size w of a machine word, and so the vectors typically consist of a small number of machine words.)

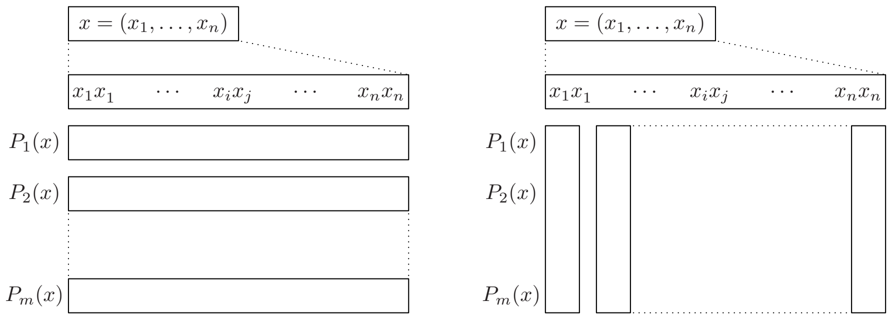


Fig. 1. Left: blocking with bitslice is too slow. Right: using lookup tables.

2.2 Computing the Polynomials

Once all monomials have been generated, one has to compute the value of every polynomial of the equation system. Recall that we are considering polynomials of the special form:

$$P_k(x_1, \ldots, x_n) = \sum_{1 \leq i \leq j \leq n} \alpha^k_{i,j} x_i x_j + \sum_{1 \leq l \leq n} \beta^k_l x_l + \gamma^k.$$

Given the value of every monomial, a straightforward computation of any polynomial would require $n(n+1)/2$ field multiplications between coefficients and monomials and the same amount of additions in order to accumulate the result. In this section, we show how to do this more efficiently depending on the context.

Blocking is not Enough. A natural way of implementing the computation of the polynomials is to perform a kind of matrix/vector product, where vector entries holds the value of the monomials and matrix rows represent the value of the polynomials' coefficients just as shown on the left side of Fig. 1. However, even when multiplications are implemented in a bitsliced fashion, this method appears to be rather slow in practice.

Lookup Tables and Field Multiplications. To compute the value of the polynomials one basically has to multiply each monomial with the corresponding coefficient in each polynomial. Of course, such a multiplication is costly and it is worth trying to avoid it; a standard way to do this is by implementing lookup tables. In our case, this strategy amounts to precomputing the contribution of a monomial to *all* polynomials simultaneously, thus saving a factor of m in the number of table lookups. This requires a lookup table with $2^p - 1$ entries of m/w machine words for all of the $n(n+1)/2$ monomials. (There is no need to store the contribution of zero, and hence there are $2^p - 1$ entries.)

Obviously, the memory required to store all these lookup tables is of crucial importance since they have to fit the processor's cache. In the special case of GF(2^4) with $n = 40$ and $m = 80$ for instance, the amount of space required is 492 K bytes, which fits the L2 cache of most processors. On the opposite, in the special case of $\mathrm{GF}(2^8)$ with $n = 20$ and $m = 40$ the required space now is strictly more than 2 M bytes and will not fit any L2 cache.

To solve this issue, it is possible to split the contribution of any monomial to the polynomials into two parts. This method may be thought of as analogous to extension towers representation of finite fields. Basically, the idea is to perform the multiplication $x \times \alpha$ as $x_{\text{low}} \times \alpha_0 \oplus x_{\text{up}} \times \alpha_1$ where x_{low} and x_{up} are respectively the $p/2$ less significant bits and the $p/2$ most significant bits and α_0 and α_1 are values of $\mathrm{GF}(2^p)$ derived from the value of the coefficient α. This very simple trick dramatically decreases the size of the lookup tables: they have $2^{p/2} - 1$ entries instead of $2^p - 1$. For instance, the previous example with $\mathrm{GF}(2^8)$, $n = 20$ and $m = 40$ now requires 252 K bytes of storage which thus fit most of current processors' L2 cache. A drawback of this technique is that the implementation becomes vulnerable to side channel attacks like cache attack [12].

2.3 The Special Case of GF(2)

There are several benefits of working in the boolean setting. Obvious remarks are that multiplications are readily handled by bitwise `and`s and that since $x_i^2 = x_i$, we only have to handle homogeneous monomials of degree two. Less obviously, the fact that a monomial now has probability $\frac{3}{4}$ to be zero leads to the optimization described in the first paragraph of this section.

Generating only the Non Zero Monomials. On the average, any variable has probability $\frac{1}{2}$ being zero. Hence any monomial of degree two is zero with probability $\frac{3}{4}$. We can take advantage of this simple fact by first computing the list of indices of non-zero variables, and then generating all pairs of such indices. The number of pairs of non-zero monomials being $n(n+1)/8$ on the average, this significantly decreases the number of lookups to the tables storing the contribution of those monomials to the polynomials and also decreases the overhead during the accumulation process. Moreover, there is no need to extract data at the bit level since all the required information can be discovered with the help of tiny auxiliary lookup tables.

A Differential Trick. To push the previous advantage one step further, the following property appears to be very useful. Every multivariate quadratic polynomial Q has the property that for any $\underline{x} = (x_1, \ldots, x_n)$ and any $\underline{\delta} = (\delta_1, \ldots, \delta_n)$, $Q(\underline{x}) \oplus Q(\underline{x} \oplus \underline{\delta}) = L_{\underline{\delta}}(\underline{x})$, where $L_{\underline{\delta}}(\underline{x})$ is linear with respect to x for any fixed value of $\underline{\delta}$. We now show how this fact can be used to amplify the cost saving achieved in the previous paragraph. Indeed, in order to compute a system $S(\underline{x})$ of multivariate quadratic equations, we first precompute the corresponding linear system $L_{\underline{\eta}}$ in $\underline{x}$ corresponding to the specific $\underline{\eta} = (1, \ldots, 1)$. For instance, the k-th row of $L_{\underline{\eta}}$ is given by:

$$P_k(\underline{x}) \oplus P_k(\underline{x} \oplus \underline{\eta}) = \sum_{1 \le i \le j \le n} \alpha_{i,j}^k \left(x_i \oplus x_j \oplus 1\right) + \text{cst}.$$

Then, depending on the weight of $\underline{x}$, we either perform the computation of $S(\underline{x})$ by evaluating $S(\underline{x})$ in case the Hamming weight of $\underline{x}$ is lower or equal to $\frac{n}{2}$, or we actually compute the mathematically equivalent function $S(\underline{x} \oplus \underline{\eta}) \oplus L_{\underline{\eta}}(\underline{x})$ in case the Hamming weight of $\underline{x}$ is bigger than $\frac{n}{2}$.

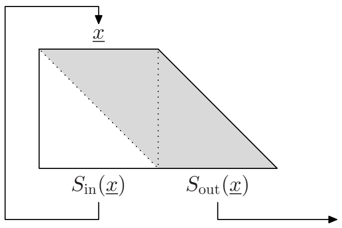
This differential trick can be pushed a little bit further with the use of an error correcting code. Considering a binary linear code, it is possible to compute $S(\underline{x})$ with $S(\underline{x} \oplus \underline{\eta}) \oplus L_{\underline{\eta}}(\underline{x})$, where $\underline{x} \oplus \underline{\eta}$ has a hamming weight lower than some value d determined by the code. However the counterpart is that we have to store for each of the code word a linear system and for each x we have to find the closest code word. For a small d, the number of code words becomes large and the memory required becomes prohibitive.

3 Some Multivariate Cryptosystems

We briefly describe in this section the multivariate schemes we implemented and for which we made extensive benchmarks on a variety of architectures.

3.1 QUAD Stream Cipher

The stream cipher QUAD is a practical stream cipher with some provable security which was introduced [2] by C. Berbain, H. Gilbert, and J. Patarin. The security proof reduces the distinguishability of the keystream generated by QUAD to the hard problem of solving randomly generated quadratic systems over finite field GF(2).


$\underline{x}$
$S_{\text{in}}(\underline{x})$
$S_{\text{out}}(\underline{x})$

The keystream generation makes use of two systems S_{in} and S_{out} of multivariate quadratic equations both sharing the same n unknowns over GF(q), as is described by the figure on the left. The first system S_{in} is used to update the internal state and thus contains n equations, whereas the second system S_{out} produces the keystream and contains $m - n$ equations. As explained in [2], the quadratic systems S_{in} and S_{out}, though randomly generated, are both publicly known.

We will restrict our study to the conservative case $m = 2n$, that is both systems S_{in} and S_{out} contain n quadratic equations in the n bits of the internal state. Given an n-bit internal state $\underline{x} = (x_1, \ldots, x_n)$, the generation amounts to iterating the following steps:

- compute $\big(S_{\text{in}}(\underline{x}), S_{\text{out}}(\underline{x})\big) = \big(Q_1(\underline{x}), \ldots, Q_{2n}(\underline{x})\big)$, from the internal state $\underline{x}$;
- output the sequence $S_{\text{out}}(\underline{x}) = \big(Q_{n+1}(\underline{x}), \ldots, Q_{2n}(\underline{x})\big)$ of n keystream bits;
- update the internal state $\underline{x}$ with the sequence $S_{\text{in}}(\underline{x}) = \big(Q_1(\underline{x}), \ldots, Q_n(\underline{x})\big)$.

The parameters recommended by the authors are $m = 2n$ and $n = 160$ over field GF(2). We made implementations for this parameters and over fields

GF(2^4) and GF(2^8). This allows us to study the impact of changing the size of the field over the performances. However there is no security arguments over fields larger than GF(2), since the security proof of [2] is only done over GF(2). In particular over GF(2^8) the number of variables becomes two small to provides a security of 2^{80}.

3.2 SFLASH Signature Scheme

The signature scheme SFLASH proposed in [1] (sometime referred to as SFLASH v2) was selected as a finalist of the NESSIE project and has resisted attacks so far. It is build around the C^* scheme of T. Matsutomo and H. Imai [7] with $\mathcal{K} = \mathrm{GF}(2^7)$ as ground field, but where some of the public equations have been removed. The secret key consists of two invertible linear transformations L_1 and L_2 defined over $\mathcal{K}^{37}$ together with an isomorphism $\varphi : \mathcal{K}^{37} \to \mathcal{L}$, where $\mathcal{L}$ is an extension of degree 37 of $\mathcal{K}$ defined by $\mathcal{L} = \mathcal{K}[y]/(y^{37} + y^{12} + y^{10} + y^2 + 1)$.

The verification algorithm recognizes $\sigma = (\sigma_1, \ldots, \sigma_{37})$ as the signature of a message $\mu = (\mu_1, \ldots, \mu_{26})$ if and only if equation

$$\mu = L_2\left(\varphi^{-1}\Big[\varphi\big[L_1(\sigma)\big]^{128^{11}+1}\Big]\right)$$

holds. Since the exponentiation $x \mapsto x^{128^{11}+1}$ is $\mathcal{K}$-quadratic, the public key which gives the values of $\mu_1, \ldots, \mu_{26}$ in terms of the variables $\sigma_1, \ldots, \sigma_{37}$ is nothing but a system of 26 multivariate quadratic equations in 37 unknowns over the finite field GF(2^7).

3.3 Rainbow Signature Scheme

Rainbow is a signature scheme proposed in [5] which is intended to rival SFLASH. However from a security point of view, Rainbow has been recently broken [3]. The public key of Rainbow consists of a set of 27 multivariate quadratic polynomials $\bar{F}_1, \ldots, \bar{F}_{27}$ in 33 unknowns over the finite field GF(2^8). The general problem of solving such a set of multivariate polynomials being hard, those polynomials are constructed in a special way using the UOV construction several times in an embedded manner to build a trapdoor.

The Rainbow signature scheme has four UOV layers and parameters $v_1 = 6$, $v_2 = 12$, $v_3 = 17$, $v_4 = 22$, and $v_5 = 33$. Each layer k is made of a as a set $\mathcal{P}_k$ of polynomials of the special form:

$$\sum_{1 \le j \le v_k < i \le v_{k+1}} \alpha_{i,j}\, x_i x_j + \sum_{1 \le i,j \le v_k} \beta_{i,j}\, x_i x_j + \sum_{1 \le i \le v_{k+1}} \gamma_i\, x_i + \delta.$$

Such polynomials are Oil and Vinegar polynomials since no monomial of degree two has both variables coming from the set $O_k = \{x_{v_k+1}, x_{v_k+2}, \ldots, x_{v_{k+1}}\}$, whereas there are monomials of degree two where both variables come from the set $V_k = \{x_1, \ldots, x_{v_k}\}$. Hence, variables from the set O_k are called oil variables of layer k, and variables from the set V_k are called vinegar variables of layer k.

The first layer of Rainbow is made of 6 polynomials randomly chosen from $\mathcal{P}_1$, the second layer is made of 5 polynomials randomly chosen from $\mathcal{P}_2$, the third layer is made of 5 polynomials randomly chosen from $\mathcal{P}_3$, and the last layer is made of 11 polynomials randomly chosen from $\mathcal{P}_4$. So the resulting internal map of Rainbow is:

$$\begin{array}{rrcl} \mathcal{F} : & \mathrm{GF}(2^8)^{33} & \longrightarrow & \mathrm{GF}(2^8)^{27}, \\ & (x_1, \ldots, x_{33}) & \longmapsto & \big(F_1(x_1, \ldots, x_{33}), \ldots, F_{27}(x_1, \ldots, x_{33})\big). \end{array}$$

Once again, the public key $\bar{\mathcal{F}}$ is obtained by applying to $\mathcal{F}$ a randomly chosen change of variables L_1 of $\mathrm{GF}(2^8)^{33}$ as well as a bijective linear output mixing L_2 of $\mathrm{GF}(2^8)^{27}$, eventually obtaining the multivariate quadratic system:

$$\bar{\mathcal{F}}(z_1, \ldots, z_{33}) = L_2 \circ \mathcal{F} \circ L_1(z_1, \ldots, z_{33}).$$

3.4 PMI$^+$ Encryption Scheme

The PMI$^+$ [4] is a doubly perturbed C^* scheme. As with C^*, there is an exponentiation $F : x \mapsto x^{2^4+1}$ defined over a finite field $\mathrm{GF}(2^{84})$, and two invertible linear transformations L_1 and L_2 respectively defined over $\mathrm{GF}(2^{84})$ and $\mathrm{GF}(2^{98})$. Let us denote by $(f_1, \ldots, f_{84})$ the binary component of the quadratic system in the 84 binary unknowns defined by F.

Additionally, randomly chose 14 quadratic polynomials $q_1, \ldots, q_{14}$ in the 84 unknowns $x_1, \ldots, x_{84}$ defined over GF(2) and a linear application Z of rank 6 from the 84 unknowns to six binary variables $z_1, \ldots, z_6$. Also randomly chose 98 quadratic polynomials $\rho_1, \ldots, \rho_{98}$ in 6 binary unknowns.

The public key is given by the expansion of the following composition:

$$L_2 \circ (f_1 + \rho_1 \circ Z, \ldots f_{84} + \rho_{84} \circ Z, \ldots q_1 + \rho_{85} \circ Z, \ldots q_{14} + \rho_1 \circ Z) \circ L_1,$$

which is a multivariate quadratic system of 98 equations in 84 unknowns defined over GF(2).

4 Implementations and Performance Results

This section exposes the benchmarks we conducted on our various implementations of the multivariate cryptosystems presented in the previous section. These benchmarks were done on several computer architectures. We used a modified version of the eSTREAM Testing Framework made by C. de Cannière [6] to evaluate the performance of our implementations when compiled with different compilers and compiling options. We mostly used compilers `gcc-4`, `gcc-3.4`, `gcc-3.3`, and `gcc-2.95`, although we also used Intel's `icc` compiler where supported. The following lists our set of machines together with a description of the processor installed:

name	**vendor**	**processor**	**frequency**	**L2 cache**
M1	Intel	Pentium 4	2505 MHz	512 kB
M2	Intel	Pentium M	1862 MHz	2048 kB
M3	Intel	Xeon	2784 MHz	512 kB
M4	AMD	Opteron	2197 MHz	1024 kB
M5	AMD	AMD64	1790 MHz	512 kB
M6	AMD	Athlon XP	2162 MHz	512 kB
M7	Power PC	G3	900 MHz	512 kB

All our implementations are written in ANSI C. Of course it is possible to improve the efficiency of these implementations by writing assembly code. However using generic code makes it possible to compare the different architectures and to evaluate cache effects.

For all versions of QUAD and PMI$^+$, speed figures are given in cycles/byte and in Mbits/second, since these are ciphers. For the SFLASH and Rainbow signature schemes, speed is given in cycles/byte and we also give the overall time needed to verify the signature.

4.1 Practical Implementations of QUAD over GF(2)

Our fastest implementation over GF(2) uses two techniques described in the previous sections: we generate only the non-zero monomials and use the differential trick. Notice that since we implement a quadratic system of 320 polynomials in 160 unknowns the total amount of storage required is about 518 K bytes so it explains the penalty on machines with 512 K bytes L2 cache.

Table 1. Speed in cycles/byte

version	M1	M2	M3	M4	M5	M6	M7
32 bit	7057	3746	4600	**2930**	3205	4866	4983
64 bit				**2081**	2636		

Table 2. Speed in Mbits/second

version	M1	M2	M3	M4	M5	M6	M7
32 bit	2.83	3.98	4.84	**6.00**	4.47	3.55	1.44
64 bit				**8.45**	5.43		

4.2 Practical Implementations of QUAD over GF(2^4)

We made five different implementations of QUAD defined over GF(2^4) with 40 unknowns and 80 polynomials. Each of these variants uses the technique of

Table 3. Speed in cycles/byte

monomials	tables	version	M1	M2	M3	M4	M5	M6	M7
naïve	2 luts	32 bit	2526	2364	2604	2134	2149	2576	**1010**
naïve	2 luts	64 bit				**1617**	1732		
naïve	1 lut	32 bit	2390	1395	1704	**1157**	1190	1546	1419
naïve	1 lut	64 bit				**994**	1639		
rotation	2 luts	32 bit	3468	2360	3452	2111	2154	2471	**977**
rotation	2 luts	64 bit				**921**	1335		
rotation	1 lut	32 bit	2858	1357	2014	**1139**	1192	1514	1435
rotation	1 lut	64 bit				**921**	1359		
bitslice	4 luts	32 bit	1906	1204	1849	1003	990	1257	**874**
bitslice	4 luts	64 bit				**745**	885		

Table 4. Speed in Mbits/second

monomials	tables	version	M1	M2	M3	M4	M5	M6	M7
naïve	2 luts	32 bit	7.93	6.30	**8.55**	8.24	6.66	6.71	7.13
naïve	2 luts	64 bit				**10.87**	8.27		
naïve	1 lut	32 bit	8.38	10.68	13.07	**15.19**	12.03	11.19	5.07
naïve	1 lut	64 bit				**17.68**	8.74		
rotation	2 luts	32 bit	5.78	6.31	6.45	**8.33**	6.65	7.00	7.37
rotation	2 luts	64 bit				**19.08**	10.73		
rotation	1 lut	32 bit	7.01	10.98	11.06	**15.43**	12.01	11.42	5.02
rotation	1 lut	64 bit				**19.08**	10.54		
bitslice	4 luts	32 bit	10.51	12.37	12.05	**17.52**	14.46	13.76	18.24
bitslice	4 luts	64 bit				**23.59**	16.18		

precomputing the contribution each monomial to all polynomials described before but with either one, two, or four tables. Implementations also have distinct monomial generation strategies.

The storage required by the coefficients of the system is about 32 K bytes. Using only one lookup table, we need to store $2^4 - 1$ times 32 K bytes, that is 492 K bytes. This value is quite close to the amount of L2 cache on some machines and thus we also implemented a version with two lookup tables. Using two lookup tables requires storing $2^2 - 1$ times 32 K bytes that is about 197 K bytes. Experimental results show that using one table is always better except on the Power PC.

For the first two implementations, we chose to generate the monomials the naïve way, and used either one or two lookup tables. For the third and fourth implementations, we used the rotation technique, and either one or two lookup tables. The last implementation, which is the fastest, uses bitslice multiplication to generate the monomials and four lookup tables of 32 K bytes, which corresponds to the contribution of each of the four bits of any monomial.

4.3 Practical Implementations of QUAD over GF(2^8)

We implemented two variants. Both of them share the monomial/coefficient multiplication precomputation technique. Since the coefficient set can definitely not be stored 255 times (it would require more than 2 M bytes), we store two lookup tables of size 25 K bytes instead.

In the first variant, we use variables rotation to generate the monomials, while in the second variant we use the naïve way to generate the monomials. Since all variables are 8-bit values, we have direct access to them which explains the fact that the naïve technique is competitive.

Table 5. Speed in cycles/byte

version	M1	M2	M3	M4	M5	M6	M7
Naive 32 bit	862	618	883	**530**	560	699	770
Naive 64 bit				**417**	464		
Rotation 32 bit	978	704	983	603	622	775	**493**
Rotation 64 bit				**497**	546		

Table 6. Speed in Mbits/second

version	M1	M2	M3	M4	M5	M6	M7
Naive 32 bit	23.25	24.10	25.22	**33.16**	25.57	24.74	9.35
Naive 64 bit				**42.15**	30.86		
Rotation 32 bit	20.49	21.16	22.66	**29.14**	23.02	22.31	14.60
Rotation 64 bit				**35.36**	26.23		

Thus, on processor M4, our implementation of QUAD over GF(2^8) achieves a throughput of 4.15 M bytes per second on the 32-bit platform and 5.27 M bytes per second on the 64-bit platform.

4.4 Practical Implementations of SFLASH

As described in the previous section, verifying an SFLASH signature can be thought of as evaluating a randomly chosen system of 26 quadratic polynomials in 37 unknowns over the finite field GF(2^7). We implemented two variants: the first one uses the naïve technique to generate the monomials, while the second one uses the rotation technique. Both of them use the precomputation of the contribution of

Table 7. Speed in cycles/byte

version	M1	M2	M3	M4	M5	M6	M7
Naive 32 bit	934	954	1046	807	848	1072	**448**
Naive 64 bit				**253**	**253**		
Rotation 32 bit	1126	863	1124	726	756	981	**452**
Rotation 64 bit				**266**	270		

Table 8. Time to verify a signature in μs

version	M1	M2	M3	M4	M5	M6	M7
Naive 32 bit	13.79	18.94	13.89	**13.58**	17.51	18.35	18.41
Naive 64 bit				**4.25**	5.22		
Rotation 32 bit	16.63	17.14	14.93	**12.22**	15.64	16.78	18.57
Rotation 64 bit				**4.47**	5.58		

the monomials to the polynomials with the help of two lookup tables. The speed measurements are obtained by verifying many signatures for the same public key.

The following table also give the overall time required to verify an SFLASH signature on the different processors:

It may be of interest to compare those figures with the `openssl` implementation of RSA-1024 and RSA-2048 signature verification. On processor M1, those implementation respectively require 2.15 ms and 3.80 ms to verify a signature. Our implementation of SFLASH on the same computer is about 150 times faster than RSA-1024.

4.5 Practical Implementations of Rainbow

Just as with SFLASH, verifying a Rainbow signature can be thought of as evaluating a randomly chosen system of 27 quadratic polynomials in 33 unknowns defined over the ground field GF(2^8). Our implementation follows the same strategy as the fastest implementation of QUAD over GF(2^8). The speed measurements are obtained by verifying many signatures for the same public key. We also give the overall time needed to verify a signature on the different processors.

Table 9. Speed in cycles/byte

version	M1	M2	M3	M4	M5	M6	M7
Naive 32 bit	1016	882	1068	719	750	989	**479**
Naive 64 bit				262	**259**		

Table 10. Time to verify a signature in μs

version	M1	M2	M3	M4	M5	M6	M7
Naive 32 bit	13.37	15.63	12.66	**10.79**	13.82	15.10	17.57
Naive 64 bit				**3.93**	4.77		

4.6 Practical Implementations of PMI$^+$

The multivariate quadratic system underlying PMI$^+$ is made of 98 polynomials in 84 unknowns over GF(2). Our implementation uses the same techniques as QUAD's implementation over GF(2), but since the numbers of variables and of polynomials are much smaller, the implementation is much faster. Additionally, the system only requires 100 K bytes of storage, so that there is no cache effect.

Table 11. Speed in cycles/byte

version	M1	M2	M3	M4	M5	M6	M7
32 bit	1443	1259	1440	909	921	1180	**589**
64 bit				**768**	821		

Table 12. Speed in Mbits/second

version	M1	M2	M3	M4	M5	M6	M7
32 bit	13.89	11.83	15.47	**19.34**	15.55	14.66	12.22
64 bit				**22.89**	17.44		

5 Conclusion

In this paper we presented several methods for efficiently implementing multivariate quadratic systems of equations. We applied these techniques to implement several multivariate cryptosystems: SFLASH, Rainbow, PMI$^+$, and the stream cipher QUAD. Our implementations were run on a large variety of architectures and appear to be quite efficient. A critical parameter when it comes to optimizations is the size of L2 cache available. Consequently, new processors with larger L2 cache leave more room for further improvement.

References

1. Akkar, M.-L., Courtois, N.T., Goubin, L., Duteuil, R.: A Fast and Secure Implementation of SFLASH. In: Desmedt, Y.G. (ed.) PKC 2003. LNCS, vol. 2567, pp. 267–278. Springer, Heidelberg (2002)
2. Berbain, C., Gilbert, H., Patarin, J.: QUAD: a Practical Stream Cipher with Provable Security. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, Springer, Heidelberg (2006)
3. Billet, O., Gilbert, H.: Cryptanalysis of Rainbow. In: De Prisco, R., Yung, M. (eds.) SCN 2006. LNCS, vol. 4116, Springer, Heidelberg (2006)
4. Ding, J., Gower, J.E.: Inoculating multivariate schemes against differential attacks. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T.G. (eds.) PKC 2006. LNCS, vol. 3958, pp. 290–301. Springer, Heidelberg (2006)
5. Ding, J., Schmidt, D.: Rainbow, a New Multivariable Polynomial Signature Scheme. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 164–175. Springer, Heidelberg (2005)
6. ECRYPT. eSTREAM: ECRYPT Stream Cipher Project, IST-2002-507932 Accessed September 29, 2005 (2005), Available at `http://www.ecrypt.eu.org/stream/`
7. Imai, H., Matsumoto, T.: Algebraic Methods for Constructing Asymmetric Cryptosystems. In: Calmet, J. (ed.) Algebraic Algorithms and Error-Correcting Codes. LNCS, vol. 229, pp. 108–119. Springer, Heidelberg (1986)
8. Kipnis, A., Patarin, J., Goubin, L.: Unbalanced Oil and Vinegar Signature Schemes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 206–222. Springer, Heidelberg (1999)

9. Matsui, M.: How Far Can We Go on the x64 Processors? In: Robshaw, M. (ed.) FSE 2006. LNCS, vol. 4047, Springer, Heidelberg (2006)
10. Matsumoto, T., Imai, H.: A Class of Asymmetric Cryptosystems Based on Polynomials over Finite Rings. In: IEEE International Symposium on Information Theory, pp. 131–132 (1983)
11. Matsumoto, T., Imai, H.: Public Quadratic Polynominal-Tuples for Efficient Signature-Verification and Message-Encryption. In: Günther, C.G. (ed.) EUROCRYPT 1988. LNCS, vol. 330, pp. 419–453. Springer, Heidelberg (1988)
12. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of aes. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)
13. Patarin, J.: Cryptoanalysis of the Matsumoto and Imai Public Key Scheme of Eurocrypt '88. In: Coppersmith, D. (ed.) CRYPTO 1995. LNCS, vol. 963, pp. 248–261. Springer, Heidelberg (1995)
14. Patarin, J.: Asymmetric Cryptography with a Hidden Monomial. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 45–60. Springer, Heidelberg (1996)
15. Patarin, J.: Hidden Fields Equations (HFE) and Isomorphisms of Polynomials (IP): Two New Families of Asymmetric Algorithms. In: Maurer, U.M. (ed.) EUROCRYPT 1996. LNCS, vol. 1070, pp. 33–48. Springer, Heidelberg (1996)
16. Patarin, J., Goubin, L., Courtois, N.T.: C*-+ and HM: Variations Around Two Schemes of T. Matsumoto and H. Imai. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 35–49. Springer, Heidelberg (1998)
17. Yang, B.-Y., Chen, J.-M., Chen, Y.-H.: TTS: High-Speed Signatures on a Low-Cost Smart Card. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, p. 371. Springer, Heidelberg (2004)