EFFICIENT PUBLIC-KEY OPERATION IN MULTIVARIATE SCHEMES

FELIPE CABARCAS*
Grupo SISTEMIC, Departamento de Ingeniería Electrónica
Universidad de Antioquia (UdeA), Calle 70 No. 52-21
Medellín, Colombia

DANIEL CABARCAS AND JOHN BAENA
Universidad Nacional de Colombia Sede Medellín
Calle 59 A N 63-20
Medellín, Colombia

(Communicated by Jintai Ding)

Abstract. The public-key operation in multivariate encryption and signature schemes evaluates \( m \) quadratic polynomials in \( n \) variables. In this paper we analyze how fast this simple operation can be made. We optimize it for different finite fields on modern architectures. We provide an objective and inherent efficiency measure of our implementations, by comparing their performance with the peak performance of the CPU. In order to provide a fair comparison for different parameter sets, we also analyze the expected security based on the algebraic attack taking into consideration the hybrid approach. We compare the attack’s efficiency for different finite fields and establish trends. We detail the role that the field equations play in the attack. We then provide a broad picture of efficiency of MQ-public-key operation against security.

1. Introduction

The quest for new hardness assumptions to support public-key cryptosystems is today more pertinent than ever. Beyond theoretical quantum algorithms that could factor integers and find discrete logarithms efficiently, recent progress in the development of quantum computers is driving governments and researchers to find alternatives. This is evident in the recently announced NIST process to standardize quantum-resistant public-key cryptographic algorithms [21].

The multivariate quadratic (MQ) problem is a plausible quantum-resistant hardness assumption [2]. Several public-key encryption [20, 22, 26, 10] and signature schemes [18, 25, 11, 23, 24, 27] have been proposed based on the hardness of the MQ problem.

Roughly, the MQ problem is the problem of solving a random system of \( m \) quadratic equations in \( n \) variables over a finite field. In MQ-based schemes, the public-key operation (encryption or signature verification) involves evaluating the
$m$ quadratic polynomials in $n$ variables. A direct attack is to solve such a quadratic system.

MQ-based schemes are appealing because the public-key operation can be made very efficient and the direct attack’s efficiency is well understood. It should be noted that several attacks [14, 19, 13, 6, 3] on specific MQ-based constructions have cast a shadow over its potential. Nevertheless some of the signature schemes have survived for a long time [18], security proofs have started to emerge [27], and new constructions seem viable [10].

1.1. OUR CONTRIBUTION. In this paper we contrast the efficiency of MQ-public-key operation against its direct attack. We consider four cases which cover most known schemes: 1) $m = n$ polynomials in $n$ variables, as in HFE [22], 2) $m = 2n$ polynomials in $n$ variables, as in ZHFE [26], 3) $m$ polynomials in $n = 3m$ variables, as in Rainbow [11], and 4) $m = n - r$ polynomials in $n$ variables, as in HFE$^-$ [24]. Cases 1) and 2) correspond to encryption schemes, while 3) and 4) correspond to signature schemes.

We optimize implementations for the MQ-public-key operation for different finite fields for an Intel x86_64 architecture with SIMD instructions. To make the code as portable as possible, without giving up performance, most of it was written without assembly-language intrinsics. In the case of GF($2^{64}$) and GF($2^{128}$) the code uses intrinsics, since there is no other way of doing it. However, for the other fields, the code was written in such a way that the compiler can vectorize it with SIMD instructions. We evaluate their efficiency comparing the performance with the peak of the computer and comparing this with a highly efficient mathematical library for floats. This provides an objective and inherent efficiency measure of our implementations. Finally, we compare the throughput for different finite fields.

It is important to notice that on a realistic application, the public-key operation is used for key exchange only, thus a single public-key operation per connection usually suffices. Our implementation and experiments reflect this scenario. We optimize the code and evaluate its performance for a single public-key operation.

We also estimate the concrete security of a prototypical MQ-based scheme. Assuming that the best attack against such a scheme is a direct algebraic attack, we estimate the efficiency of the Groebner basis computation. In some cases the hybrid approach is the best attack, i.e., guessing a number of variables and doing the Groebner basis computation thereafter. Our analysis considers the optimal number of variables to guess. We compare the attack’s efficiency for different finite fields and establish trends for very large fields.

We perform an analysis to have a better understanding of the role that the field equations play in the direct attack, which allows us to include such field equations in the cases in which they enhance the performance of the attack. The so called field equations ($x_i^2 - x_i = 0$) are known to speed up the direct attack for the field of order two. However, their effect is not well-documented for other fields. We show that when solving $2n$ equations in $n$ variables they play an important role for GF($2$) and GF($3$), and have no effect for fields of order greater than 12. When solving $n$ equations in $n$ variables, they play a role if the field order is smaller than $n$. In addition, and of independent interest, we analyze the optimal number of variables to guess when field equations are included in the Groebner basis computation.

Finally, we bring together public-key-operation data and attack data to provide a realistic picture of the scheme’s efficiency. Interestingly, the best performers are mid-size prime fields. This is despite the natural representation of GF($2$) and the...
very suitable carry-less multiplication instruction PCLMULQDQ for GF(2) extensions.

1.2. RELATED WORK. Chen et al. [9] provide a broad discussion on how to use modern CPUs to implement various MQ-cryptographic algorithms. In fact, they favor the matrix-vector multiplication approach for the public-key operation, as we do, and discuss which intrinsics can be used to speed up this operation. Similarly, Berbain et al. [1] discuss various implementation strategies focusing on the cases GF(2), GF(2⁴) and GF(2⁸). We build upon their work, using some of the techniques they propose, and extending to other parameters when necessary. We note that some of the techniques proposed in [1], intended for the QUAD stream cipher, are impractical for public-key operation because they require to construct large lookup tables, which are a burden for a single operation. To show the applicability of the proposed implementations, we made CPU performance analysis. Also, we include security estimates that contribute for a fairer comparison between different parameter sets. It is also important to mention that by the time of the publication of [1] and [9], the PCLMULQDQ instruction was not available, which makes a big difference today on large GF(2) extensions.

1.3. ORGANIZATION. This article is organized as follows. In Sect. 2, we describe our implementations of MQ-public-key operations and also present results obtained with those implementations. In Sect. 3 we discuss a security analysis of some MQ schemes, and in Sect. 4, based on the analysis of the previous sections, we present some tools and ideas to choose optimal parameters for MQ schemes. We finalize giving some conclusions in Sect. 5.

2. OPTIMIZING MQ-PUBLIC-KEY OPERATION

In this section, we present implementations of MQ-public-key operation in C++ for different finite fields and we study their performance. We consider the four cases of variables and equations mentioned in Section 1.1. Aiming at a wide range of field sizes, we include a much more ample range than schemes commonly used, yet making sure that commonly used field sizes are covered by similar sizes in our analysis. In order to evaluate the efficiency of the proposed implementations, we measure their throughput and we analyze their efficiency comparing the performance with the peak of the computer and comparing this with a highly efficient mathematical library for floats. This provides an objective and inherent efficiency measure of our implementations.

The MQ-public-key operation consists on evaluating \( m \) quadratic polynomials in \( n \) variables. This can be efficiently implemented as a matrix-vector multiplication [9]. The matrix consists of the coefficients of the polynomials, while the vector is built from the variables \((x_1, \ldots, x_n)\) as

\[
(x_1^2, x_1 x_2, \ldots, x_1 x_n, x_2^2, x_2 x_3, \ldots, x_2 x_n, \ldots, x_{n-1} x_n, x_n^2, x_n, 1).
\]

Notice that the vector has \( l = \frac{n(n+1)}{2} + n + 1 \) entries and the matrix is \( m \times l \).

The key matrices were stored in column-major order to exploit parameter locality. This way, the coefficients of the polynomials of the same monomial, are consecutive in memory. The implementations loop first over the polynomials and then over the
monomials, providing a large number of independent operations for the processor, and efficient use of loads of the cache lines.

2.1. Computer and Compiler Details. The computer used to test the performance is a Desktop PC with an Intel® Core™ i5-2310 CPU @ 2.90GHz processor (2nd Gen Intel® Core™ Processor Family Desktop, microarchitecture codename “Sandy Bridge”). Its cache hierarchy is 32KB Data L1, 256KB L2 and 8MB L3. Like any modern processor, it has Out of Order (OOO) scheduling execution. We only describe integer instructions since the algorithms operate over finite fields. The processor ISA contains the PCLMULQDQ instruction (carry-less multiplication), which is important for one of the implementations presented. This processor can execute 3 integer instructions per cycle or 2 vector operation (multiplication and addition) per cycle [17]. The integer vector registers are 128 bits, therefore, it can execute 4 single precision (32 bits) or 2 double precision (64 bits) integer instructions per cycle.

The code was written in C++, and it was compiled using gcc 4.8.4. The optimization flags used were: -mpclmul -O3 -mavx; but the required flags depend on the prime field code, as it will be explained below. The Operating System used was Ubuntu (version 14.04). Most of the code was written without assembler intrinsics; they were used only when the field order was a power of two, which required the carry-less multiplication. Therefore, the vectorization (the use of SIMD instructions) was left for compiler automatic optimization. The purpose of this was to make the code as portable as possible without giving up performance. Apart from the intrinsics used, mentioned before, the code was carefully written to ease the compiler optimization; e.g., in the prime field code, the inner loop of the dot product was manually unrolled 4 times.

2.2. Finite Field Operations. The MQ-public-key operation requires finite field’s additions and multiplications. The implementation of these operations differ drastically depending on the finite field used. In prime finite fields, additions and multiplications can be implemented using integer additions and multiplications, respectively, followed by a reduction operation. These can be computed with a modulo operation or \( \mod \) operation. In the case of binary fields, polynomial operations are required. Different techniques were used to implement them.

2.2.1. Odd prime field: In order to implement the multiplication over a prime field of order \( q \), it has to be determined how to use the primitive instructions of the processor. Processors represent integers in binary with a fixed number of digits. Most of the time they are interpreted as unsigned or signed integers (the latter ones represented as two’s complement). We chose unsigned integers for the implementations. For a given number of bits, multiplication can generate numbers twice as big, and the processors usually provide flags and registers that can be used to determine the exact result and if there was overflow. However, in order to determine if there was overflow, it is necessary to add conditionals, which increase complexity and reduce efficiency considerably. Since the objective of this work is to create a fast implementation of the operation mechanism, we decided to avoid as many conditional branches as possible. Therefore we had to make sure that multiplications will not overflow the data type used to encode the message.

If the size of the prime field is smaller than \( \sqrt{M} \), with \( M \) being the largest integer that can be represented in a specific integer representation, each multiplication and addition can be implemented using: two computer instructions, the operation and
the reduction (with the mod instruction). However, if the prime field is small enough compared with \( M \), it is possible to make a single reduction at the end of each dot product.

In the single reduction strategy, the largest integer that can be obtained before reduction happens when all the key values are the maximum of the prime field \( q \), i.e. \( q - 1 \), and the message is also composed of only the maximum. This is not a likely key, but it is an upper bound. Then, the maximum value that will be obtained in each dot product is

\[
\frac{n(n+1)}{2}(q-1)^3 + n(q-1)^2 + (q-1).
\]

The first term corresponds to the quadratic part of the polynomials, there are \( n(n+1)/2 \) such monomials, and assuming that the components of the vector (1) and the coefficients are all \( q - 1 \), we have \( (n(n+1)/2)(q-1)^3 \) in the worst case. Similarly, the second and last terms represent the linear and constant parts of the polynomials, respectively. Therefore, from equation (2), the possibility of using a single reduction strategy depends on \( q \), \( n \), and the maximum value of the integer representation. E.g., if we set \( n = 150 \), we can use this strategy, in the case of 32 bits for integers up to \( q = 73 \), while in the case of 64 bits, the largest finite field is \( q = 117659 \).

In comparison with a naive implementation, which requires \( l \) multiplications and additions, and \( 2l \) modular reductions, since a reduction takes about 10 times more cycles than an addition or a multiplication, we are reducing by a factor of 11\( l \) the number of cycles required.

In the simulations, we assume that the keys are stored in the corresponding Field (one key per int) and the message is stored in bit string. So the messages are converted to the corresponding prime field taking 32 bits at the time, dividing by the field as many times as required.

2.2.2. \( GF(2) \): Multiplications and additions on the \( GF(2) \) can be implemented using logical AND and XOR respectively (no need for reduction). Most computers’ ISA contain these logical bitwise instructions, so the use of 64-bit (unsigned long long) data type allows to multiply or add 64 elements at a time. Sandy Bridge microarchitecture contains 256-bit vector registers, that could be used to multiply simultaneously up to 256 elements. Even though there are no bitwise logical AND and XOR for the integer type in the vector unit used, these bitwise instructions exist for floating point operands. The compiler produces the bitwise floating point instructions \( \text{vandps} \) and \( \text{vxorps} \) for the integer type \( \text{unsigned long long} \). The Listing 1 shows the C code for the matrix times vector operation, while Listing 2 shows the assembler code generated. It can be seen in the assembly code that there is no penalty on converting the integer to float, since the compiler knows that it can not convert the type and it is not needed.

In Listing 1, the number of words required (\( nWR \)) is the number of equations divided by 256 (number of bits of the largest vector instruction of the processor). Similarly, the \( \text{messageVec}[4] \) is actually a single position of the input vector. The value of each of the input vector is replicated to 256 bits (4 consecutive memory positions of 64 bits each); this way, the compiler can generate vector instructions that compute 256 points in a single computer instruction. The code is written like this in order to avoid the use of intrinsic (assembly) instructions. Note that even if the processor does not include vector instructions, the code is correct (but probably
slower). It is important to explain here that the input vector (in a variable `vector` in Listing 1) is stored in a single bit per each position of the array, and the bit is replicated in all 64 bits of each.

Similarly to the prime field implementation, a naive implementation in GF(2) requires $l$ multiplications and additions, and $2l$ modular reductions. Since we are using registers that store 256 elements each, and use a single instruction AND to multiply and a single OR instruction to add, then, 2 instructions replace 256 multiplication, 256 additions and 512 reductions.

**Listing 1.** Matrix times vector GF(2).

```c
for (int i = 0; i < polySize; i++) {
    // \textit{nWR is the number of words required}
    unsigned long long mVec[4];
    mVec[0] = vector[i];
    mVec[1] = vector[i];
    mVec[2] = vector[i];
    mVec[3] = vector[i];

    for (int k = 0; k < nWR; k+=1) {
        unsigned long long prod[4];

        prod[0] = keyCoef[i*nWR+k][0] & mVec[0];
        prod[1] = keyCoef[i*nWR+k][1] & mVec[1];
        prod[2] = keyCoef[i*nWR+k][2] & mVec[2];
        prod[3] = keyCoef[i*nWR+k][3] & mVec[3];

        mOut[k][0] = mOut[k][0] ^ prod[0];
        mOut[k][1] = mOut[k][1] ^ prod[1];
        mOut[k][2] = mOut[k][2] ^ prod[2];
        mOut[k][3] = mOut[k][3] ^ prod[3];
    }
}
```

**Listing 2.** Inner loop of the matrix times vector operation as generated by the compiler.

```
e8 : vmovdqu (%rcx,%rax,1),%xmm1
ed : vinsertf128 $0x1,0x10(%rcx,%rax,1),%ymm1,%ymm1
f5 : vmovdqu (%rdx,%rax,1),%xmm0
fa : vinsertf128 $0x1,0x10(%rdx,%rax,1),%ymm0,%ymm0
102: vandps %ymm2,%ymm1,%ymm1
106: vxorps %ymm0,%ymm1,%ymm0
10a : vmovdqu %ymm0 (%rdx,%rax,1)
10f : vextractf128 $0x1,%ymm0
117: add $0x20,%rax
```

Advances in Mathematics of Communications, Volume 13, No. 2 (2019), 343–371
2.2.3. \(GF(2)\) extension: In this case, each multiplication is a polynomial operation, therefore, a naive implementation will be very slow compared with the ones presented in previous sections. In this implementation an integer would hold as many coefficients as possible of each polynomial. The addition operation could be implemented as a logical XOR of two integers. However, multiplication will be very demanding since it would require, at least, a loop the size of the field extension; and another loop would be required for the reduction operation. In order to improve this we have modified the code presented on Intel’s white Paper [15]. The paper shows how to use the carry-less multiplication instruction PCLMULQDQ to multiply two polynomials in GF(2), therefore removing the loops of a naive implementation.

The PCLMULQDQ instruction multiplies two degree 64 polynomials with coefficients in GF(2). Line 9 of Listing 3 multiplies the polynomials \(a\) and \(b\), while lines 10 to 13 reduce the multiplication module the polynomial \(g\). This reduction, as explained in [15], is an extension of the Feldmeier CRC generation algorithm. Furthermore, the implementation of 128 bits (Listing 4) is similar, but it requires more operations given the size of the polynomials and restrictions of the PCLMULQDQ instruction that only multiplies polynomials of degree 64.

We have made some modifications, on this code, to allow any reduction polynomial, and the use both 64 and 128 coefficients. In our implementation, 21 operations are required for each multiplication (with its respective reduction) in the case of 128 extension field, shown in Listing 4, while 5 operations are required in the case of 64 extension field, shown in Listing 3.

In a naive implementation of GF(2) Extension, each multiplication and reduction requires polynomial operations. Furthermore, a multiplication of degree 64 polynomials would require \(64 \cdot 64\) multiplications and \(64\) additions. In the proposed implementation, a single instruction replaces all these operations. And a polynomial division would require at least \(64\) polynomial subtractions and the same number of conditionals to test results, and in the implementation all those are replaced by 4 processor instructions. Therefore the proposed implementation is several times faster than a naive one.

**Listing 3. Galois field multiplication polynomials \(2^{64}\)**

```c
// res = a * b % poly
inline void Zhs64::gfmul (_m128i& a, _m128i& b, _m128i& res) {
    _m128i g = poly;
    _m128i C, D, Ar, Er;

    C = _mm_clmulepi64_si128(a, b, 0x00);
    Ar = _mm_clmulepi64_si128(C, g, 0x01);
    D = _mm_xor_si128( Ar, C);
    Er = _mm_clmulepi64_si128(D, g, 0x01);
    C = _mm_xor_si128( Er, C);
    res = C;
}
```

```c
11 b: cmp %rsi,%rax
11 e: jne e8
```
Listing 4. Galois field multiplication polynomials $2^{128}$.

```c
// res = a ∗ b % poly
inline void Zhefe128::gfmul (_m128i& a, _m128i& b, _m128i& res) {
    _m128i g = poly;
    _m128i C, D, E, F, H;
    _m128i Ar, Br, Cr;
    _m128i Er, Fr, Hr;

    C = _mm_clmulepi64_si128(a, b, 0x00);
    E = _mm_clmulepi64_si128(a, b, 0x10);
    F = _mm_clmulepi64_si128(a, b, 0x01);
    D = _mm_clmulepi64_si128(a, b, 0x11);

    F = _mm_xor_si128(F, E);
    E = _mm_slli_si128(F, 8);
    F = _mm_srli_si128(F, 8);
    C = _mm_xor_si128(C, E);
    D = _mm_xor_si128(D, F);

    Ar = _mm_clmulepi64_si128(D, g, 0x10);
    Br = _mm_clmulepi64_si128(D, g, 0x01);
    Cr = _mm_clmulepi64_si128(D, g, 0x11);
    Ar = _mm_xor_si128(Ar, Br);

    Ar = _mm_srli_si128(Ar, 8);
    Ar = _mm_xor_si128(Ar, Cr);
    D = _mm_xor_si128(Ar, D);

    Er = _mm_clmulepi64_si128(D, g, 0x00);
    Fr = _mm_clmulepi64_si128(D, g, 0x01);
    Fr = _mm_slli_si128(Fr, 8);
    Hr = _mm_xor_si128(Er, Fr);

    H = _mm_xor_si128(Hr, C);

    res = H;
}
```

2.3. THROUGHPUT RESULTS. Here we present results about the performance of MQ-public-key operation. We measure the throughput of encryption (cases 1 and 2) in bytes-per-microsecond and that of signature verification (cases 3 and 4) in signatures-per-second.

2.3.1. Encryption. Figures 1 and 2 show throughput, in bytes-per-microsecond, for cases 1) and 2) versus the number of variables, for different finite fields. Each point on the graph is the median time to encrypt a 5000 messages (because the characteristics of the computer used, the average was not a good statistic for time to encrypt). Furthermore, each message was encrypted making sure the key was not present on the L3 cache.
For all fields, as the number of variables increase, the throughput decreases. This is because the number of operations increase squared with the number of variables, while the size of the message increases only linearly, therefore, the efficiency (the throughput) is reduced squared.

In the case of prime fields, for a fixed computer type, as the prime field increases the throughput also increases; e.g. for 32-bit integers, with $n = 60$, the public-key operation for GF(73) has a larger throughput than for GF(13). Since the number of instructions required for these two prime fields encodings depends on the computer type, not on the field itself, they all require the same number of computer instructions, but information processed by each processor instruction depends on the prime field and as it increases, more information is processed, increasing the throughput.

It can be seen that the throughput for GF($2^{64}$) is greater than for GF($2^{128}$). Even though the number of multiplications and additions of the polynomials depend on the computer type (which are the same for both), and the amount of bytes processed in each operation is twice for GF($2^{128}$) field, the number of computer operations required for each multiplication is more than twice for GF($2^{128}$) than for GF($2^{64}$). The effect is that the field GF($2^{64}$) has a higher throughput.
Figure 2. Public-key-operation throughput for different number of variables for $m = 2n$. The register type used in the program to hold the message is fixed for each finite field and shown in brackets. See Figure 1 for details.

Finally, GF(2) has the slowest decrease of all, so it becomes the best (in throughput) for large number of variables for both $m = 2n$ and $m = n$. The slowest decrease is because this field has the most efficient implementation. In this field, there is no need to divide by the module after multiplications or additions and several operations can be made with a single instruction, since bitwise instructions implement multiplication and addition directly in the field. We used 256 bit registers on the implementation, so each computer multiplication or addition processes up to 256 encryption operations; the downside of this can be seen at low number of variables, where there is a low efficiency compared to the other fields, this if due to a waste on operations, since the registers used are too big for the size of the problem.

2.3.2. Signature verification. Figures 3 and 4 show throughput of cases 3) and 4) in signatures-per-second using similar conditions as for encryption. Just like encryption, for signature verification the throughput decreases as $n$ increases. Unlike encryption, for prime fields, for a fixed computer type, the throughput stays constant as the prime field increases. This is because the amount of information processed does not matter, since we only count number of signatures. For the same reason compounded with the advantage of GF($2^{64}$) exposed above, the throughput for field GF($2^{64}$) is much higher than that of GF($2^{128}$). Similarly, it is very natural that GF(2) has the highest throughput.

2.4. Percentage of the computer peak performance. A common way to assess the efficiency of an algorithm’s implementation is to measure how close to
Figure 3. Public-key-operation throughput for different number of variables in the case \( n = 3m \). The register type used in the program to hold the message is fixed for each finite field and shown in brackets. See Figure 1 for details.

peak performance of the computer is achieved. In modern personal computers, the memory hierarchy, and the speed of the memory are the main factors that reduce the efficiency of a program. In order to get close to the peak performance of a processor, it is usually required that there are many operations for each memory load, since the processor speed is usually so high that it can perform hundreds or thousands of ALU operations for each memory load or store that has to go to main memory. In the case of the algorithm required for the MQ-public-key operation, the problem is that there is little data reuse possible, since the key matrix values are used only one time. Therefore it is not possible to get close to peak, unless the whole matrix is stored in L1 cache at the beginning of the algorithm, and in the experiments we made sure that the key is not in cache at the beginning of each operation. In this way we simulate the use of public-key operation by a user that just processes a single message.

In order to estimate the implementation’s efficiency, we calculate the peak performance of the algorithm based on the instructions and the characteristics of the processor, to get an expected throughput. And, based on the throughput of the program, we get the fraction of peak obtained, as shown in Figure 5. Furthermore, as a reference for the peak performance that can be achieved by the processor we have chosen a mathematical operation that has been greatly studied and optimized, but it is also similar to the operation that has to be implemented. This operation is the matrix times a vector, and the implementation selected was the Atlas’ Sgemv...
Figure 4. Public-key-operation throughput for different number of variables in the case \( m = n - r \) with \( r = 16 \). The register type used in the program to hold the message is fixed for each finite field and shown in brackets. See Figure 1 for details.

[30], which makes the operation over float numbers (32 bits). We measure this implementation peak performance for the same matrix size as the ones we are working for in the case of the public-key operation.

The number of operations required to multiply a matrix of size \( m \) by \( l \) and a vector of size \( l \) are: \( ml \) multiplications and \( ml \) additions. Additionally, \( \frac{n(n+1)}{2} \) multiplications are required to create the vector. In this paper we are considering four different cases, however, for the peak performance we analyze only the case of \( m = 2n \) equations in \( n \) variables. In this case \( l = \frac{n(n+1)}{2} + n + 1 \) a total of

\[
n^3 + 7n^2/2 + 5n/2
\]

multiplications and

\[
n^3 + 3n^2 + 2n
\]

additions are required to encrypt each message of size \( n \). The number of computer instructions required, depends on the field used, and it is analyzed below.

For public-key operation over a prime field, the computer peak performance, in Bytes/\( \mu s \), can be calculated as

\[
\frac{n \times \log_2 q \text{bytes} \times 2900 \text{cycles} \mu s \times X \text{inst. cycle}}{8} \times \frac{1}{2n^3 + 13n^2/2 + 9n/2} \times \frac{1}{\text{inst.}},
\]
where $X$ depends on the integer type used. In the case of 16 bits \textit{(unsigned short)} the vector instructions (256-bit registers) of integer multiplication can do 8 multiplications in a single instruction ($vpmullw$) and a vector addition ($vpaddw$) at the same time, therefore ($X = 16$). While in the case of 32 bits \textit{(unsigned int)} the instruction ($vpmulld$) can do 4 multiplications in a single instruction and 4 additions with the $vpaddd$ instruction for ($X = 8$) in this case.

For 64-bit type \textit{(unsigned long long)} there is no vector instruction available, however, the architecture has 3 ALU execution units, so $X = 3$. It is important to mention that when the 64-bit arithmetic was used, the best results were obtained with the option \texttt{-fno-tree-vectorize"}. The problem is that \texttt{gcc}'s options -O3 and -mavx try to use vector instructions, and since there are no 64-bit-integer-vector operations, the compiler converts the 64-bit multiplications in several 32-bit multiplications plus move and additions instructions, which end up being less efficient that a scalar integer multiplication.

In the case of public-key operation over GF(2), the computer peak performance, in Bytes/$\mu$s, for this algorithm can be calculated as

\[
\frac{n}{8} \text{bytes} \times \frac{2900 \text{cycles}}{\mu s} \times \frac{1 \text{inst.}}{\text{cycle}} \times \frac{1}{(n^3 + 3n^2 + 2n)/128 + n(n + 1)/2} \times \frac{1}{\text{inst.}}.
\]

In the case of GF(2) extension, the computer peak performance, in Bytes/$\mu$s, for this algorithm can be calculated as
3. Concrete security of some MQ systems

The public key in an MQ scheme is a quadratic map $GF(q)^n \rightarrow GF(q)^m$. A direct attack consists in solving a system of $m$ quadratic equations in $n$ variables over $GF(q)$. These schemes are usually constructed with the aim that the best attack is a direct attack and that such attack is no more efficient than solving a set of random quadratic equations of the same size [26, 28, 10, 12, 18]. Although seldom achieved, there are important exceptions such as UOV [18] and Rainbow [11]. In most cases, specific attacks have rendered the schemes insecure.

In this section we establish the concrete security of such schemes based on the direct algebraic attack for different parameters. By doing so we do not claim that the direct attack is the best attack against any specific scheme. Rather, we aim at establishing a common security measure to compare the efficiency of different parameter sets of MQ schemes.

Previous work discussing the efficiency of a direct attack only took into consideration the number of equations and variables. Their authors neglected the effect of the size of the field probably because it is small in comparison. Because we want to find the optimal parameters for some MQ systems, here we dig deeper into these matters. We assume that such an attack behaves as solving random quadratic equations, thus it grows exponentially in $n$. All the experiments in this section were performed using Magma v2.21-1 [7] on a server with a processor Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz, running Linux CentOS release 6.6.

We consider four cases which cover most known schemes: 1) $m = n$ polynomials in $n$ variables, 2) $m = 2n$ polynomials in $n$ variables, 3) $m$ polynomials in $n = 3m$ variables, and 4) $m = n - r$ polynomials in $n$ variables. Cases 1) and 2) correspond to typical encryption schemes, while 3) and 4) correspond to signature schemes.

When dealing with square semiregular sequences, there is a technique that, for some parameters, can enhance the performance of the direct algebraic attack on such equations. This method is known as the “hybrid approach” [4, 5], and the way it works is that before we compute the Groebner basis of the system, we want to randomly guess $k$ of the $n$ variables in order to obtain an overdetermined system, and hope that this action produces a smaller total time for the attack. Notice that the total time for the hybrid approach will be the time of the direct attack with $n - k$ variables times $q^k$, which is the number of times that we would have to perform the direct algebraic attack for each set of parameters.

An important point in the hybrid approach is determining the optimal number of guessed variables $k_0$. Bettale, Faugère, and Perret observed in [5] that $k_0$ is a multiple of $n$, $k_0 = \beta_0 n$ and provide a way to find $\beta_0$ in Proposition 3.2. Table 1 shows the best trade-off to solve a square system using this approach. The approach in [5] does not take into account the field equations\(^1\). For this case, we follow the

\[^1\text{Hereafter the term } wfe \text{ will stand for "with field equations".}\]
algorithmic approach proposed in [4] to determine the optimal trade-off for a specific value of \( n \). The results are showed in Figure 6. It can be observed that there is considerable variation. For example, with \( q = 3 \), for \( 30 \leq n \leq 120 \), \( \beta_0 \) varies between 0.43 and 0.60. However, the complexity is quite stable in that range. In order to approximate the asymptotic behaviour, we chose an arbitrary value for \( \beta \) in the optimal range, and calculated Groebner bases time.

| \( q \) | 2 | 2(wfe) | 3 | 3(wfe) | 13 | 13(wfe) | 73 | 919 | 117659 |
|-------|---|--------|---|--------|---|---------|---|----|-------|
| \( \beta_0 \) | 0.76 | 0.55 | 0.61 | 0.50 | 0.27 | 0.23 | 0.12 | 0.05 | 0.02 |

Table 1. Optimal fraction of variable to guess in hybrid attack.

![Figure 6. Optimal fraction of variable to guess in hybrid attack with field equations.](image)

We have used the hybrid approach with the optimal \( k_0 \) obtained according to Table 1 for cases 1), 3) and 4), for all values of \( q \), except for \( q = 2^{64} \) and \( q = 2^{128} \), for which the size of the field do not allow the hybrid approach to be of much help. Also, our results show that for \( q = 13 \) the plain algebraic attack – when \( k = 0 \) – is slightly better than the hybrid approach, so the latter is not considered either for this field size. For case 2) we have not used the hybrid attack, since the equations already form a very overdetermined system, which defeats the purpose of such approach.

3.1. Case 1: \( m = n \). In this section we consider the MQ systems in which the number of polynomials and the number of variables are the same. Figure 7 shows the time in seconds to attack an MQ scheme as \( n \) increases for selected field sizes...
\( q \), using the direct/hybrid attack\(^2\). We observe exponential trends, with different growth rates. The trend line equation is of the form
\[
\text{time in seconds} = c e^{bn},
\]
where \( n \) is the number of variables and \( c \) and \( b \) are as in Table 2.

![Figure 7](image.png)

\textbf{Figure 7.} Time to attack for different values of \( n \) and selected fields, in the case \( m = n \) equations in \( n \) variables.

Notice that there are two sets of points for GF(2), GF(3) and GF(13), one with and the other without field equations. It is well understood that when \( q \in \{2, 3, 13\} \), and since we know there is a solution in the ground field, as part of the computation we can include the so called field equations \( x_i^q - x_i \). By doing so, we force the Groebner basis algorithm to look for a solution in GF(\( q \)) as opposed to do it in the algebraic closure. This makes the algorithm considerably more efficient.

We know that for semiregular sequences of \( n \) quadratic equations in \( n \) variables, for any \( q \) and without the use of field equations, the degree of regularity is \( n + 1 \). Thus, if we choose an \( n \) larger than \( q \), then the field equations would have an impact on the performance of the direct attack. This fact must be taken into account when choosing the different parameters for an MQ scheme. Therefore, for \( q \in \{2, 3, 13\} \), we have included data obtained with and without the field equations \( x_i^q - x_i \).

For \( q = 2 \) there is yet another alternative for attacking a square generic quadratic system. Bouillaguet et al. pushed the exhaustive search to the limit by using properties of differentials, gray codes and careful optimization [8]. We took into account this approach and found out that on the computer architecture we are

\(^2k=0\) for \( q = 13 \), \( q = 2^{64} \) and \( q = 2^{128} \).
considering, it is slightly better than the hybrid approach for \( n < 100 \) but worst thereafter. Therefore we exclude these results. It is important to mention though that this approach is highly parallelizable and thus suitable for GPUs.

3.2. Case 2: \( m = 2n \). In this section we consider the MQ systems in which the number of polynomials is twice the number of variables. Figure 8 shows the time in seconds to attack an MQ scheme as \( n \) increases for selected field sizes \( q \). Although the exponential trend is apparent for every field size, the growth rate is considerably different. The trend line equation is of the form

\[
\text{time in seconds} = c e^{bn},
\]

where \( n \) is the number of variables and \( c \) and \( b \) are as in Table 3.

| \( q \)   | \( c \)         | \( b \)     |
|----------|----------------|-------------|
| 2        | \( 1.43 \times 10^{-4} \) | 0.61        |
| 2 wfe    | \( 1.58 \times 10^{-5} \) | 0.58        |
| 3        | \( 8.62 \times 10^{-6} \) | 0.95        |
| 3 wfe    | \( 2.55 \times 10^{-6} \) | 0.98        |
| 13       | \( 2.32 \times 10^{-6} \) | 1.53        |
| 13 wfe   | \( 3.16 \times 10^{-8} \) | 1.35        |
| 73       | \( 1.30 \times 10^{-6} \) | 1.76        |
| 919      | \( 2.73 \times 10^{-5} \) | 1.63        |
| 117659   | \( 2.50 \times 10^{-8} \) | 1.66        |
| \( 2^{38} \) | \( 3.97 \times 10^{-7} \) | 2.27        |
| \( 2^{128} \) | \( 1.20 \times 10^{-6} \) | 2.18        |

Table 2. Trend line parameters, in the case \( m = n \) equations in \( n \) variables.

| \( q \)   | \( c \)         | \( b \)     |
|----------|----------------|-------------|
| 2        | \( 3.97 \times 10^{-6} \) | 0.74        |
| 2 wfe    | \( 3.92 \times 10^{-6} \) | 0.54        |
| 3        | \( 5.47 \times 10^{-9} \) | 0.99        |
| 3 wfe    | \( 1.99 \times 10^{-8} \) | 0.88        |
| 13       | \( 5.37 \times 10^{-1} \) | 1.00        |
| 73       | \( 4.68 \times 10^{-7} \) | 1.01        |
| 919      | \( 5.04 \times 10^{-7} \) | 1.00        |
| 117659   | \( 4.06 \times 10^{-7} \) | 1.02        |
| \( 2^{38} \) | \( 1.40 \times 10^{-6} \) | 0.90        |
| \( 2^{128} \) | \( 2.00 \times 10^{-9} \) | 1.04        |

Table 3. Trend line parameters, in the case \( m = 2n \) equations in \( n \) variables.

For the cases \( q = 5 \) and \( q = 7 \), due to time and memory limitations, the values of \( n \) that we could reach were not high enough for the field equations to start being helpful for the algebraic attack (the degree of regularity achieved was too low). However, considering the trend observed for the obtained degrees of regularity, we can expect that at a certain point, for practical values of \( n \), the field equations will start to enhance the attack for these fields. As a result, the data obtained for these cases should not be used to construct trend lines as we did for other values of \( q \).
Reporting this data could lead the reader to mistakenly consider that information to extrapolate to higher values of \( n \). Nevertheless, we would like to add that if someone is interested in studying the cases \( q = 5 \) and \( q = 7 \), then they should count with enough time –perhaps months– and memory to reach values of \( n \) for which the degree of regularity is sufficiently greater than \( q \). Our experience tells us that considering larger values than 5 or 7 for \( q \) usually ends up in better security levels against the algebraic attack. In summary, we think that leaving \( q = 5 \) and \( q = 7 \) out of the picture do not represent any significant loss of information.

An MQ scheme can be constructed over any odd prime field. It is interesting to observe the behaviour of the attack as the field size increases. Figure 9 depicts the time to attack an MQ scheme with \( n = 16 \) and \( n = 17 \) variables (32 and 34 equations respectively) as the prime size field increases. The uneven behaviour of the graph is due to the different representations of primes used by the Magma software. After a jumpy start, the graph tends to flatten after \( \log(q) = 33 \). After this point, time increases linearly in the number of bits of the representation. However, one can expect further jumps down the road. The uneven behaviour is very similar for \( n = 16 \) and \( n = 17 \).

Attack time increases more evenly for GF(2) extensions. Figure 10 shows the time to attack an MQ scheme with \( n = 16 \) variables and 32 equations for growing GF(2) extensions. In this case time is a low degree (around 2) polynomial function of the number of bits of the representation. There is however a considerable jump from \( q = 2^{32} \) to \( q = 2^{31} \), where the time almost doubles. These is likely to be due to a change in the use of processor intrinsics from 32 to 64 bit commands.
In Figure 11 we present the degree of regularity of semiregular sequences of $2n$ quadratic equations in $n$ variables, for any $q$ and without the use of field equations. We observe that, for $n \leq 80$, the degree of regularity is no greater than 12. Thus, if we chose $q > 12$, then the field equations would not have any impact whatsoever on the performance of the direct attack.

3.3. Case 3: $n = 3m$. We now consider the MQ signature systems in which the number of variables is 3 times the number of polynomials. This setup includes the UOV signature scheme [18]. Since the number of variables exceeds the number of polynomials by $2m$, in order to speed up the attack, we randomly guess $2m$ of those variables before running the Groebner basis algorithm. If the solution set that we obtain is empty, we repeat this process until we find a solution for the equations. We performed extensive experiments for different sets of parameters and found that, on average, less than two trials were needed to obtain a solution for each system of equations. With this approach, the time and degree of regularity needed to find a solution for a system with $m$ equations in $3m$ variables, were essentially the same that the ones needed to obtain a solution for a system with $m$ equations in $m$ variables. In other words, this case is included in case 1), keeping in mind that, on average, the Groebner basis algorithm must be run at most twice to forge a signature, for all values of $q$ and $m$. Recall that in case 1) we have used the hybrid approach for all values of $q$, except for $q = 13$, $q = 2^{64}$ and $q = 2^{128}$. Therefore, we have done the same for case 3), i.e., we have used the hybrid attack for the same parameters. Figure 12 shows the time in seconds to attack an MQ scheme as $n$ increases for selected field sizes $q$. 

**Figure 9.** Time to attack $n = 16$ and $n = 17$, for different prime fields, in the case $m = 2n$ equations in $n$ variables.
3.4. Case 4: $m = n - r$. We now consider the MQ signature systems in which the number of polynomials is $r$ units less than the number of variables. As in case 3), since the number of variables exceeds the number of polynomials by $r$, we randomly guess $r$ of those variables before running the Groebner basis algorithm. This process must be repeated until we find a solution for the equations. We performed numerous experiments for different sets of parameters, and found that, on average, at most two trials were needed to obtain a solution for each system of equations. Using this technique, the time and degree of regularity needed to find a solution for a system with $n - r$ equations in $n$ variables, were essentially the same that the ones needed to obtain a solution for a system with $n - r$ equations in $n - r$ variables. As in case 3), we conclude that this case is included in case 1), taking into account that, on average, the Groebner basis algorithm must be run at most twice to forge a signature, for all values of $q$, $n$ and $r$. As in cases 1) and 3), we have used the hybrid approach for all values of $q$, except for $q = 13$, $q = 2^{64}$ and $q = 2^{128}$. Figure 13 shows the time in seconds to attack an MQ scheme as $n$ increases for selected field sizes $q$ and $r = 16$.

3.5. Conversion to bit security. From the total time of an attack, we then compute a bit security level $b$, following the methodology of Howgrave-Graham [16]. Assuming that a single block-cipher encryption takes 500 clock cycles, it would take $2^b \cdot \frac{500}{2.4 \times 10^9}$ seconds to attack a $b$-bit block-cipher using brute-force on a 2.40GHz machine. From this, we obtain that the bit security of a cryptosystem, that can be attacked in no less than $T$ seconds, is given by

$$b = \log_2(T) + \log_2 \left(2.4 \times 10^9\right) - \log_2(500).$$
Tables 4, 5, 6, and 7 show the number of variables needed to achieve different bit security levels, for distinct field sizes. Once the field size and the security level are established, we then can use these tables to choose the proper number of variables to be used.

It is worth pointing at a peculiar phenomenon in Tables 4, 6 and 7. The monotonicity of $n$ with respect to field size is broken for the largest fields. As $n$ increases, the time to attack is dominated by Groebner basis symbolic computations and the field size becomes less relevant, thus, the gap between the time to attack for different fields reduces. As a consequence, the larger gap for small values of $n$, artificially reduces the slope of the trendline for larger fields. One should really think that the time to attack converges as $n$ increases for large enough fields.

In [29], Thomae and Wolf present a method to get better results than simply guessing $2^m$ variables in the case when $n = 3m$. They prove that with a linear change of variables, the complexity of solving a system with $m$ equations in $n = 3m$ variables can be reduced to the complexity of solving a determined system with $m - 2$ equations and variables. Therefore, after determining the tentative values of $n$ for Table 6, each of them must be increased by 6 units in order to take into account this improvement.

Another important issue in the signature cases is the collision attack against the underlying hash function. In cases 3) and 4) the scheme signs a message of $m$ GF($q$) elements. Assuming that a $k$ bit hash provides $k/2$ bits security, and taking into consideration the encoding of GF($q$) elements explained in Section 2, $n \geq$...
Figure 12. Time to attack for different values of $n$ and selected fields, in the case $m$ equations in $n = 3m$ variables.

| $q$    | bit sec. level | 2  | 3  | 13 | 73 | 919 | 107659 | $2^{64}$ | $2^{128}$ |
|--------|----------------|----|----|----|----|-----|--------|----------|-----------|
| 80     | 88             | 54 | 38 | 31 | 32 | 28  | 25     | 25       |
| 100    | 112            | 68 | 48 | 39 | 40 | 36  | 31     | 31       |
| 112    | 126            | 77 | 54 | 43 | 45 | 41  | 34     | 35       |
| 128    | 145            | 88 | 62 | 50 | 52 | 48  | 39     | 40       |
| 192    | 222            | 133| 95 | 75 | 79 | 75  | 59     | 61       |

Table 4. $n$ needed for different bit security levels, when $m = n$. For $q = 13, 2^{64}, 2^{128}$ the best attack is the plain algebraic attack, and for the rest is the hybrid approach.

$6b[32 \log_q(2)]/32$ provides $b$ bits of security in case 3) and $n \geq 2b[32 \log_q(2)]/32 + r$ in case 4). These lead to better attacks when $q = 2, 3, 13$ but not for $q \geq 73$. Tables 6 and 7 reflect this.

4. Optimal parameters

A realistic picture of the scheme’s efficiency only emerges when we take into consideration security. Figures 14, 15, 16, and 17 depict public-key-operation throughput against security. The throughput data comes from the measures explained in Section 2 and the security is estimated based on the analysis in Section 3. In the
Figure 13. Time to attack for different values of $n$ and selected fields, in the case $n - r$ equations in $n$ variables, for $r = 16$.

| $q$     | 2  | 3  | 13 | 73 | 919 | 117659 | $2^{64}$ | $2^{128}$ |
|---------|----|----|----|----|-----|--------|----------|-----------|
| bit sec. level |  80 |  97 |  61 |  55 |  55 |  54  |  51    |  49    |
|          | 100 | 123 |  77 |  69 |  69 |  68  |  65    |  63    |
|          | 112 | 138 |  86 |  77 |  77 |  76  |  73    |  71    |
|          | 128 | 158 |  99 |  88 |  88 |  87  |  84    |  81    |
|          | 192 | 240 | 150 | 133 | 132 | 130  | 127    | 124    |

Table 5. $n$ needed for different bit security levels, when $m = 2n$.

following subsections we discusses the results for encryption (cases 1 and 2) and signature verification (cases 3 and 4).

4.1. Encryption. The best performance for encryption measured in bytes-per-microsecond is achieved by GF(117659) encoded as 64 bit unsigned integer. This is an example of using integer instructions optimally to avoid modular reduction. It shows the effectiveness of this simple technique, which allows a very good use of the processor ISA. In second place we have GF(73) encoded as 32 bit integer, which is again an example of the same technique. The fact that GF(117659) is more efficient than GF(73) is due to the fact that, on modern processor, the microarchitecture can
execute 64 bit integer instructions almost as fast as a 32 bits integer instructions. It is important to note that the 32 bit integer instructions were executed in vector units. Notice that this technique cannot be easily used on the Groebner basis attack, where the number of operations is much larger and unknown.

Next in performance come GF(919) and GF(13). There is a significant waste as these are encoded as 32- and 64-bit integers, which can allocate larger fields. Both have very similar throughput if \( m = 2n \), but GF(919) clearly outperforms GF(13) when \( m = n \) due to the effect of field equations.

The effect of field equations is even more acute for GF(2) and GF(3). GF(2) looks like the most efficient when plotted against number of variables (see Figures 1 and 2). In reality, when plotted against security, it has an intermediate performance in the case \( m = 2n \), and it is the worst for \( m = n \). This is due to the effectiveness of the use of field equations in Groebner basis computations, which significantly speeds up the attack.

GF\( (2^{64}) \) and GF\( (2^{128}) \) are not as bad as one might expect. Despite the complexity of multiplication in these fields, they have a competitive performance in the case

| bit sec. level | \( q \) |
|---------------|-------|
|               | 2     | 3    | 13   | 73   | 919  | 117659 | \( 2^{64} \) | \( 2^{128} \) |
| 80            | 480   | 315  | 135  | 98   | 100  | 89     | 81         | 82         |
| 100           | 600   | 394  | 169  | 121  | 125  | 114    | 100        | 102        |
| 112           | 672   | 441  | 189  | 135  | 141  | 129    | 111        | 113        |
| 128           | 768   | 504  | 216  | 154  | 161  | 149    | 126        | 129        |
| 192           | 1152  | 756  | 324  | 230  | 243  | 229    | 187        | 192        |

Table 6. \( n \) needed for different bit security levels, when \( n = 3m \).

For \( q = 2, 3, 13 \) the best attack is a collision attack on the underlying hash function, for \( q = 73, 919, 117659 \) is the hybrid approach, and for the rest is the plain algebraic attack.

| bit sec. level | \( q \) |
|---------------|-------|
|               | 2     | 3    | 13   | 73   | 919  | 117659 | \( 2^{64} \) | \( 2^{128} \) |
| 80            | 176   | 121  | 61   | 47   | 48   | 44     | 41         | 42         |
| 100           | 216   | 147  | 72   | 55   | 56   | 52     | 48         | 48         |
| 112           | 240   | 163  | 79   | 59   | 61   | 57     | 51         | 52         |
| 128           | 272   | 184  | 88   | 66   | 68   | 64     | 56         | 57         |
| 192           | 400   | 268  | 124  | 91   | 95   | 91     | 77         | 78         |

Table 7. \( n \) needed for different bit security levels, when \( m = n-r \), for \( r = 16 \). For \( q = 2, 3, 13 \) the best attack is a collision attack on the underlying hash function, for \( q = 73, 919, 117659 \) is the hybrid approach, and for the rest is the plain algebraic attack.
$m = n$. This highlights the effectiveness of the carry-less multiplication instruction PCLMULQDQ, in particular in the case of $\text{GF}(2^{64})$ where multiplication can be done in 5 operations. Moreover, they are not effected by the hybrid approach.

4.2. Signature verification. The picture of efficiency for signature verification measured in signatures-per-second is quite different (see Figures 16 and 17). The best performance is achieved by $\text{GF}(73)$. Here, using integer instructions optimally to avoid modular reduction plays only a minor role. Since the amount of information processed does not matter, the larger fields are penalized. However, for $\text{GF}(2)$, $\text{GF}(3)$ and $\text{GF}(13)$, the field equations are very effective in speeding up the attack, thus these small fields are less efficient than $\text{GF}(73)$ where the field equations do no play a role. $\text{GF}(2)$, $\text{GF}(3)$ and $\text{GF}(13)$ are further penalized because of the possibility of a collision attack on the underlying hash function. In a sense, the amount of processed information does matter, when it is below certain threshold, it makes the scheme susceptible to a collision attack.

Interestingly, $\text{GF}(2^{64})$ and $\text{GF}(3)$ have a very similar performance. Here, we observe a balance between the penalty to large fields due to waste of information and the penalty to small fields due to a collision attack.

5. Conclusions

We have proposed efficient implementations of MQ-public-key operations in C++, for different finite fields and set ups. We have optimized these implementations exploiting modern computer architectures.
Figure 15. Public-key-operation throughput for different security levels in the case $m = 2n$ equations in $n$ variables.

Figure 16. Public-key-operation throughput for different security levels in the case $m$ equations in $3m$ variables.

We have obtained a very detailed security analysis of the direct attack against MQ-schemes, and that has given us a more complete understanding of this attack.
Eff. P-K operation in multivar. schemes

Figure 17. Public-key-operation throughput for different security levels in the case $m = n - r$ equations in $n$ variables with $r = 16$.

For instance, we have carefully established when the field equations should be added in order to enhance the performance of the attack.

Combining results of public-key operation and security, we have built a graph of Public-key-operation throughput for different security levels. This graph provides a realistic picture of the scheme’s efficiency. According to this graph, the best performers are surprisingly mid-size prime fields and not the field GF(2), despite the natural representation of the field GF(2) and the very suitable carry-less multiplication instruction PCLMULQDQ for its extensions.

We have also built a very useful tool to choose optimal parameters for MQ schemes, when considering their security against the direct attack.

Acknowledgments

We would like to thank Jintai Ding, Daniel Escudero, Javier Verbel and Jaiberth Porras for useful discussions. We would also like to thank the reviewers of this paper for their constructive reviews and suggestions. And last but not least, we thank the Facultad de Ciencias of the Universidad Nacional de Colombia sede Medellín for granting us access to the Enlace server, where we ran most of the experiments of this paper.

References

[1] C. Berbain, O. Billet and H. Gilbert, Efficient implementations of multivariate quadratic systems, in Selected Areas in Cryptography (eds. E. Biham and A. Youssef), vol. 4356 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, 174–187.
[2] D. J. Bernstein, J. Buchmann and E. Dahmen, Post-Quantum Cryptography, Springer-Verlag, Berlin, 2009.
[3] L. Bettale, J.-C. Faugère and L. Perret, Cryptanalysis of the TRMS Signature Scheme of PKC’05, Progress in cryptology – AFRICACRYPT 2008, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, 143–155.
[4] L. Bettale, J.-C. Faugère and L. Perret, Hybrid approach for solving multivariate systems over finite fields, Journal of Mathematical Cryptology, 3 (2009), 177–197.

[5] L. Bettale, J.-C. Faugère and L. Perret, Solving polynomial systems over finite fields: Improved analysis of the hybrid approach, in ISSAC 2012 - 37th International Symposium on Symbolic and Algebraic Computation, ACM, Grenoble, France, 2012, 67–74.

[6] L. Bettale, J.-C. Faugère and L. Perret, Cryptanalysis of HFE, multi-HFE and variants for odd and even characteristic, Designs, Codes and Cryptography, 69 (2013), 1–52.

[7] W. Bosma, J. Cannon and C. Playoust, The Magma algebra system. I. The user language, J. Symbolic Comput., 24 (1997), 235–265.

[8] C. Bouillaguet, H.-C. Chen, C.-M. Cheng, T. Chou, R. Niederhagen, A. Shamir and B.-Y. Yang, Fast exhaustive search for polynomial systems in $F_2$, in Cryptographic Hardware and Embedded Systems, CHES 2010 (eds. S. Mangard and F.-X. Standaert), Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, 203–218.

[9] A.-T. Chen, M.-S. Chen, T.-R. Chen, C.-M. Cheng, J. Ding, E.-H. Kuo, F.-S. Lee and B.-Y. Yang, See implementation of multivariate pkcs on modern x86 cpus, in Cryptographic Hardware and Embedded Systems - CHES 2009 (eds. C. Clavier and K. Gaj), vol. 5747 of Lecture Notes in Computer Science, Springer Computer Science, Springer Berlin Heidelberg, 2009, 33–48.

[10] J. Ding, A. Petzoldt and L.-c. Wang, The cubic simple matrix encryption scheme, in Post-Quantum Cryptography: 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings (ed. M. Mosca), Springer International Publishing, Cham, 2014, 76–87.

[11] J. Ding and D. Schmidt, Rainbow, a new multivariable polynomial signature scheme, in Applied Cryptography and Network Security (eds. J. Ioannidis, A. Keromytis and M. Yung), vol. 3531 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, 164–175.

[12] J. Ding, D. Schmidt and F. Werner, Algebraic attack on HFE revisited, in Information Security, 11th International Conference, ISC 2008, Taipei, Taiwan, September 15-18, 2008. Proceedings (eds. T.-C. Wu, C.-L. Lei, V. Rijmen and D.-T. Lee), vol. 5222 of Lecture Notes in Computer Science, Springer, 2008, 215–227.

[13] V. Dubois, P.-A. Fouque, A. Shamir and J. Stern, Practical cryptanalysis of Sflash, in CRYPTO, 4622 (2007), 1–12.

[14] J.-C. Faugère and A. Jouy, Algebraic cryptanalysis of hidden field equation (HFE) cryptosystems using Gröbner bases, in Advances in Cryptology—CRYPTO 2003, vol. 2729 of Lecture Notes in Comput. Sci., Springer, Berlin, 2003, 44–60.

[15] S. Gueron and M. E. Kounavis, White Paper: Intel Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode, Technical report, Intel, 2010.

[16] N. Howgrave-Graham, A hybrid lattice-reduction and meet-in-the-middle attack against ntru, in Advances in Cryptology - CRYPTO 2007 (ed. A. Menezes), vol. 4622 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, 150–169.

[17] Intel®, Intel®64 and IA-32 Architecture Optimization Reference Manual, Technical report, Intel®, 2015.

[18] A. Kipnis, J. Patarin and L. Goubin, Unbalanced oil and vinegar signature schemes, in Advances in Cryptology—EUROCRYPT ’99 (Prague), vol. 1592 of Lecture Notes in Comput. Sci., Springer, Berlin, 1999, 206–222.

[19] A. Kipnis and A. Shamir, Cryptanalysis of the HFE public key cryptosystem by relinearization, in Advances in Cryptology—CRYPTO ’99 (Santa Barbara, CA), vol. 1666 of Lecture Notes in Comput. Sci., Springer, Berlin, 1999, 19–30.

[20] T. Matsumoto and H. Imai, Public quadratic polynomial-tuples for efficient signature-verification and message-encryption, in Advances in Cryptology—EUROCRYPT ’88 (Davos, 1988), vol. 330 of Lecture Notes in Comput. Sci., Springer, Berlin, 1988, 419–453.

[21] NIST, Proposed submission requirements and evaluation criteria for the post-quantum cryptography standardization process, http://csrc.nist.gov/groups/ST/post-quantum-crypto/call-for-proposals-2016.html, 2016, Accessed: 08/26/2016.

[22] J. Patarin, Hidden Field Equations (HFE) and Isomorphisms of Polynomials (IP): Two new families of asymmetric algorithms, in Advances in Cryptology—EUROCRYPT 96 (ed. U. Maurer), vol. 1070 of Lecture Notes in Computer Science, Springer-Verlag, 1996, 33–48.

[23] J. Patarin, N. Courtois and L. Goubin, FLASH, a fast multivariate signature algorithm, in Topics in Cryptology—CT-RSA 2001 (San Francisco, CA), vol. 2020 of Lecture Notes in Comput. Sci., Springer, Berlin, 2001, 298–307.
[24] J. Patarin, N. Courtois and L. Goubin, QUARTZ, 128-bit long digital signatures, in Topics in cryptology—CT-RSA 2001 (San Francisco, CA), vol. 2020 of Lecture Notes in Comput. Sci., Springer, Berlin, 2001, 282–297.

[25] J. Patarin, L. Goubin and N. Courtois, C*-+ and HM: Variations around two schemes of T. Matsumoto and H. Imai, in ASIACRYPT ’98: Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security, Springer-Verlag, London, UK, 1998, 35–50.

[26] J. Porras, J. Baena and J. Ding, ZHFE, a new multivariate public key encryption scheme, in Post-Quantum Cryptography (ed. M. Mosca), vol. 8772 of Lecture Notes in Computer Science, Springer International Publishing, 2014, 229–245.

[27] K. Sakumoto, T. Shirai and H. Hiwatari, Public-key identification schemes based on multivariate quadratic polynomials, in Advances in Cryptology - CRYPTO 2011 (ed. P. Rogaway), vol. 6841 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2011, 706–723.

[28] C. Tao, A. Diene, S. Tang and J. Ding, Simple matrix scheme for encryption, in Post-Quantum Cryptography: 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings (ed. P. Gaborit), Springer Berlin Heidelberg, Berlin, Heidelberg, 6841 (2013), 231–242.

[29] E. Thomae and C. Wolf, Solving underdetermined systems of multivariate quadratic equations revisited, in Public Key Cryptography – PKC 2012 (eds. M. Fischlin, J. Buchmann and M. Manulis), 7293 (2012), 156–171.

[30] R. C. Whaley and A. Petitet, Minimizing development and maintenance costs in supporting persistently optimized BLAS, Software: Practice and Experience, 35 (2005), 101–121.

Received July 2018; revised January 2019.
E-mail address: felipe.cabarcas@udea.edu.co
E-mail address: dcabarc@unal.edu.co
E-mail address: jbbaena@unal.edu.co