SQUARING ALGORITHMS WITH DELAYED CARRY METHOD AND EFFICIENT PARALLELIZATION

Kovtun Vladislav, Okhrimenko Andrew

Increasing amounts of information that needs to be protecting put in claims specific requirements for information security systems. The main goal of this paper is to find ways to increase performance of cryptographic transformation with public key by increasing performance of integers squaring. Authors use delayed carry mechanism and approaches of effective parallelization for Comba multiplication algorithm, which was previously proposing by authors. They use the idea of carries accumulation by addition products of multiplying the relevant machine words in columns. As a result, it became possible to perform addition of such products in the column independently of each other. However, independent accumulation of products and carries require correction of the intermediate results to account for the accumulated carries. Due to the independence of accumulation in the columns, it became possible to parallelize the process of products accumulation that allowed formulating several approaches. In this paper received theoretical estimates of the computational complexity for proposed squaring algorithms. Software implementations of algorithms in C++ allowed receiving practical results of the performance, which are not contrary to theoretical estimates. The authors first proposed applying the method of delayed carry and parallelization techniques for squaring algorithms, which was previously proposing for integer multiplication.

Keywords: squaring, multiplication, integers, delayed carry, parallelization.

1 Introduction

Cryptographic transformation with public key (CTPK) are the basis for most modern cryptosystems. Increasing amounts of information that needs to be protected, makes specific demands for CTPK. Multiplicative operations (Denis and Rose, 2006), (Hankerson, Menezes, Vanstone, 2004), such as multiplication and squaring of integers, are the most frequently used in CTPK. One of the performance increasing approaches in CTPK is increasing the productivity of basic operations, such as multiplication, squaring, modular reduction and multiplicative inversion. Performance increasing approaches in CTPK by increasing the productivity of integer multiplication were reviewed in (Kovtun, Okhrimenko, Nechiporuk, 2012), (Kovtun and Okhrimenko, 2012), (Kovtun and Okhrimenko, 2013). The **main goal** of this paper is to find ways of increasing performance of CTPK, by increasing productivity of squaring integers, using the delayed carry mechanism (Kovtun et al., 2012), (Kovtun and Okhrimenko, 2013) and efficient parallelization approaches (Kovtun and Okhrimenko, 2012).

Squaring is a special case of multiplication where both multipliers are equal (Handbook, 2006), (Hankerson et al., 2004). Show features of multiplication and squaring by considering "schoolbook" multiplication of two integers 123 and 456, Fig. 1:

		1	2	3	
×		4	5	6	
		$6 \cdot 1$	$6 \cdot 2$	$6 \cdot 3$	Row 0
	$5 \cdot 1$	$5 \cdot 2$	$5 \cdot 3$		Row 1
$4 \cdot 1$	$4 \cdot 2$	$4 \cdot 3$			Row 2
Column 4	Column 3	Column 2	Column 1	Column 0	

Fig. 1. "Schoolbook" multiplication of two integers

Fig.1 shows that to calculate the product of two integers 123 and 456, it should complete 9 unique multiplication operations. Squaring using "schoolbook" multiplication allows some optimizations. Multiply integer 123 by itself, using "schoolbook" multiplication Fig. 2.

		1	2	3	
×		1	2	3	
		$3 \cdot 1$	$3 \cdot 2$	$3 \cdot 3$	Row 0
	$2 \cdot 1$	$2 \cdot 2$	$2 \cdot 3$		Row 1
$1 \cdot 1$	$1 \cdot 2$	$1 \cdot 3$			Row 2
Column 4	Column 3	Column 2	Column 1	Column 0	

Fig. 2. "Schoolbook" multiplication integer by itself

Fig. 2 shows how to multiply decimals in a different order, such as products in rows 0 and 2 in column 2 (**3 · 1 = 1 · 3),** products in rows 0 and 1 in column 1 (**3 · 2 = 2 · 3)** and products in rows 1 and 2 in column 3: (**1 · 2 = 2 · 1)**. Therefore, for squaring for n-digit number, there are only $\left(n^2+n\right)/2$ unique multiplications required (n^2 operations required for multiplication in common case).

Let x be integer being squared, a x_k – k*-th* term of x. It is easy to notice features:

1. In row k the product in column $2k$ has a x_k^2 term in it. In Fig. 2 it **3 · 3, 2 · 2, 1 · 1**.
2. Every non-square term of a column will appear twice (product in column j in row k, where $j \neq 2k$ has a pair). In Fig. 2 it **3 · 1 = 1 · 3, 3 · 2 = 2 · 3** and **1 · 2 = 2 · 1**. Every odd column is made-up entirely of product pairs.
3. For row k, such as $k \neq 0$ and $k \neq n-1$, the first unique product that is not a square, is located in the column $2k+1$. In Fig. 2 it **2 · 1**.

2 Multiplication algorithm Modified Comba

In (Kovtun and Okhrimenko, 2013) proposed generalized modified algorithm ***Comba*** for integer multiplication – ***Modified Comba (MC)***, which uses the idea of delayed carry. The basis of the algorithm is loops (p.2 and p.3), and inner loops (p 2.1 and p 3.1). At the lowest level of the hierarchy, in loops p. 2.1, p. 3.1 there are multiplication and accumulation of delayed carry. Accumulated carry is taken into account in the final iterations of the loops p. 2 and p. 3. Using *2w*-bit variables for storing *w*-bit variables eliminates the carry accounting of *w*-bit variable after each arithmetic operation. Carry accumulated in the higher part of the 2w-bit variable and is taken into account when needed, Fig. 3. The generalized algorithm ***MC*** (Kovtun and Okhrimenko, 2013) for the w-bit systems is given below.

Multiplication algorithm 1. Modified Comba

INPUT	**OUTPUT**
$a, b \in \mathbf{GF}(p)$, $n = \log_{2^w} a$, $nk = 2n-1$	$c = a \cdot b$

1. $r_0^{(2w)} \leftarrow 0$, $r_1^{(2w)} \leftarrow 0$, $r_2^{(2w)} \leftarrow 0$.

2. For $k \leftarrow 0$, $k < n$, $k{+}{+}$ do

2.1. For $i \leftarrow 0$, $j \leftarrow k$, $i \le k$, $i{+}{+}$, $j{-}{-}$ do

2.1.1. $(uv)^{(2w)} \leftarrow a_i^{(w)} \cdot b_j^{(w)}$.

2.1.2. $r_0^{(2w)} \leftarrow r_0^{(2w)} + v^{(w)}$, $r_1^{(2w)} \leftarrow r_1^{(2w)} + u^{(w)}$ ***// delayed carry accumulation***

2.2. $r_1^{(2w)} \leftarrow r_1^{(2w)} + \mathrm{hi}_{(w)}\left(r_0^{(2w)}\right)$, $r_2^{(2w)} \leftarrow r_2^{(2w)} + \mathrm{hi}_{(w)}\left(r_1^{(2w)}\right)$ ***// delayed carry accounting***

2.3. $c_k^{(w)} \leftarrow \mathrm{low}_{(w)}\left(r_0^{(2w)}\right)$, $r_0^{(2w)} \leftarrow \mathrm{low}_{(w)}\left(r_1^{(2w)}\right)$, $r_1^{(2w)} \leftarrow \mathrm{low}_{(w)}\left(r_2^{(2w)}\right)$, $r_2^{(2w)} \leftarrow 0$.

3. For $k \leftarrow n$, $l \leftarrow 1, k < nk$, $k{+}{+}$, $l{+}{+}$ do

3.1. For $i \leftarrow l$, $j \leftarrow k - l$, $i < n$, $i{+}{+}$, $j{-}{-}$ do

3.1.1. $(uv)^{(2w)} \leftarrow a_i^{(w)} \cdot b_j^{(w)}$.

3.1.2. $r_0^{(2w)} \leftarrow r_0^{(2w)} + v^{(w)}$, $r_1^{(2w)} \leftarrow r_1^{(2w)} + u^{(w)}$ ***// delayed carry accumulation***

3.2. $r_1^{(2w)} \leftarrow r_1^{(2w)} + \mathrm{hi}_{(w)}\left(r_0^{(2w)}\right)$, $r_2^{(2w)} \leftarrow r_2^{(2w)} + \mathrm{hi}_{(2)}\left(r_1^{(2w)}\right)$ ***// delayed carry accounting***

3.3. $c_k^{(w)} \leftarrow \mathrm{low}_{(w)}\left(r_0^{(2w)}\right)$, $r_0^{(2w)} \leftarrow \mathrm{low}_{(w)}\left(r_1^{(2w)}\right)$, $r_1^{(2w)} \leftarrow \mathrm{low}_{(w)}\left(r_2^{(2w)}\right)$, $r_2^{(2w)} \leftarrow 0$.

4. $c_{nk}^{(w)} \leftarrow \mathrm{low}_{(2)}\left(r_0^{(2w)}\right)$.

5. Return (c).

The computational complexity of the ***MC*** algorithm:

$$I_{sqr}^{MC} = n^2\left(I_{mul}^{w} + \left(\frac{3n+1}{n}\right)I_{add}^{2w+w}\right),$$

where n – number of w-bit machine words required to store the multiplier of given size, I_{mul}^{w} – a multiplication operation for w-bit words, I_{add}^{2w+w} – an addition operation for $2w$-bit and w-bit words. Assignment operations do not take into account in computational complexity of the algorithms.

Using the idea of delayed carry it can independently produce addition of multiplication results corresponding by columns, that enables to perform the accumulation of sum of high and least significant bit in separate parallel threads. However, it is necessary to make an adjustment (account carry) $r_1 = r_1 + \mathrm{Hi}(r_0)$, $r_2 = r_2 + \mathrm{Hi}(r_1)$ and set result $c_i = \mathrm{Low}(r_0)$ after sum accumulation in each thread. Fig. 3 and Fig. 4 is a graphical interpretation of the ***MC*** algorithm, for n=3, where well-defined results addition for corresponding products in columns.

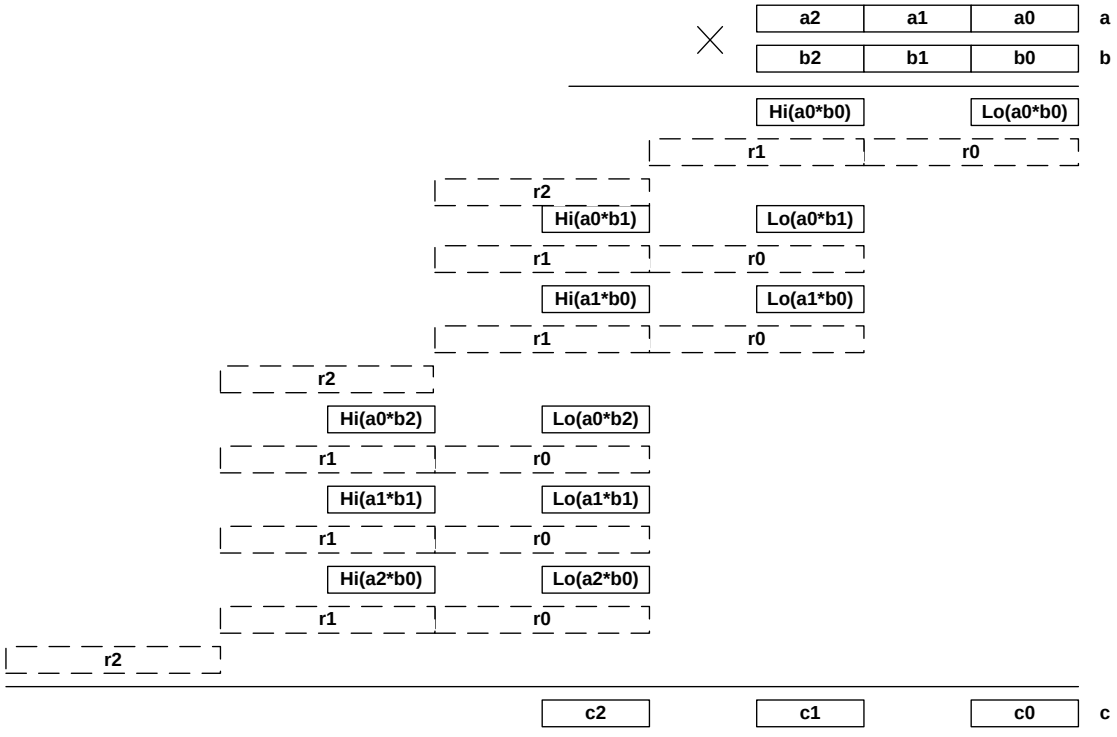

Fig. 3. Graphical interpretation of loop 2 in ***MC*** algorithm

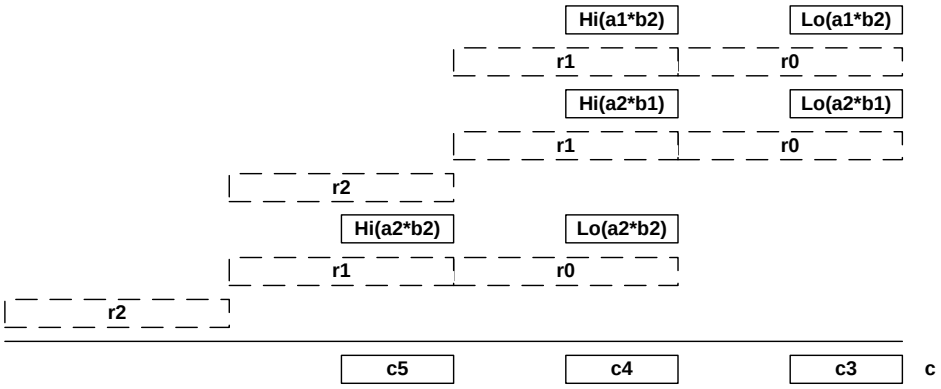

Рис. 4. Graphical interpretation of loop 3 in ***MC*** algorithm

3 SQUARING ALGORITHMS

3.1 Squaring algorithm Modified Comba SQR

Using delayed carry mechanism (Kovtun et al., 2012), (Kovtun and Okhrimenko, 2013) and approaches to parallelization (Kovtun and Okhrimenko, 2012), (Kovtun and Okhrimenko, 2013), were offer three squaring algorithms that take account the above features. Consider squaring features, the ***MC*** algorithm was modified in inner loops of delayed carry accumulation (p.2.1 and p.3.1), and added an additional check to avoid duplication in the sum accumulation (p.2.1.2 and p.3.1.2).

Modified Comba SQR (MCSQR) is squaring algorithm for *w*-bit machine words, based on multiplication algorithm ***MC*** (Kovtun et al., 2012), (Kovtun and Okhrimenko, 2012), (Kovtun and Okhrimenko, 2013).

Squaring algorithm 1. Modified Comba SQR

INPUT	**OUTPUT**
$a \in \mathrm{GF}(p),\ n = \log_{2^w} a,\ nk = 2n-1$	$c = a^2$

1. $r_0^{(2w)} \leftarrow 0,\ r_1^{(2w)} \leftarrow 0,\ r_2^{(2w)} \leftarrow 0.$

2. For $k \leftarrow 0,\ k < n,\ k{+}{+}$ do

2.1. For $i \leftarrow 0,\ j \leftarrow k,\ i \le j,\ i{+}{+},\ j{-}{-}$ do

2.1.1. $\left(u^2\right)^{(2w)} \leftarrow a_i^{(w)} \cdot a_j^{(w)}.$

2.1.2. if $(i < j)$ then $r_0^{(2w)} \leftarrow r_0^{(2w)} + \left(u^{(w)} << 1\right),$
$r_1^{(2w)} \leftarrow r_1^{(2w)} + \left(\left(u^{(w)} << 1\right) \ or\ \left(u^{(w)} >> (w-1)\right)\right);$
else $r_0^{(2w)} \leftarrow r_0^{(2w)} + u^{(w)},\ r_1^{(2w)} \leftarrow r_1^{(2w)} + u^{(w)}$

} ***delayed carry accumulation***

2.2. $r_1^{(2w)} \leftarrow r_1^{(2w)} + \mathrm{hi}_{(w)}\left(r_0^{(2w)}\right),\ r_2^{(2w)} \leftarrow r_2^{(2w)} + \mathrm{hi}_{(w)}\left(r_1^{(2w)}\right)$ // ***delayed carry accounting***

2.3. $c_k^{(w)} \leftarrow \mathrm{low}_{(w)}\left(r_0^{(2w)}\right),\ r_0^{(2w)} \leftarrow \mathrm{low}_{(w)}\left(r_1^{(2w)}\right),\ r_1^{(2w)} \leftarrow \mathrm{low}_{(w)}\left(r_2^{(2w)}\right),\ r_2^{(2w)} \leftarrow 0.$

3. For $k \leftarrow n,\ l \leftarrow 1, k < nk,\ k{+}{+},\ l{+}{+}$ do

3.1. For $i \leftarrow l,\ j \leftarrow k - l,\ i \le j,\ i{+}{+},\ j{-}{-}$ do

3.1.1. $\left(u^2\right)^{(2w)} \leftarrow a_i^{(w)} \cdot a_j^{(w)}.$

3.1.2. if $(i < j)$ then $r_0^{(2w)} \leftarrow r_0^{(2w)} + \left(u^{(w)} << 1\right),$
$r_1^{(2w)} \leftarrow r_1^{(2w)} + \left(\left(u^{(w)} << 1\right) \ or \ \left(u^{(w)} >> (w-1)\right)\right);$
else $r_0^{(2w)} \leftarrow r_0^{(2w)} + u^{(w)},\ r_1^{(2w)} \leftarrow r_1^{(2w)} + u^{(w)}$
} ***delayed carry accumulation***

3.2. $r_1^{(2w)} \leftarrow r_1^{(2w)} + \mathrm{hi}_{(w)}\left(r_0^{(2w)}\right),\ r_2^{(2w)} \leftarrow r_2^{(2w)} + \mathrm{hi}_{(2)}\left(r_1^{(2w)}\right)$ // ***delayed carry accounting***

3.3. $c_k^{(w)} \leftarrow \mathrm{low}_{(w)}\left(r_0^{(2w)}\right),\ r_0^{(2w)} \leftarrow \mathrm{low}_{(w)}\left(r_1^{(2w)}\right),\ r_1^{(2w)} \leftarrow \mathrm{low}_{(w)}\left(r_2^{(2w)}\right),\ r_2^{(2w)} \leftarrow 0.$

4. $c_{nk}^{(w)} \leftarrow \mathrm{low}_{(2)}\left(r_0^{(2w)}\right).$

5. Return (c).

The computational complexity of the ***MCSQR*** algorithm:

$$I_{sqr}^{MCSQR} = n^2\left(I_{mul}^{w} + \left(\frac{3n+1}{n}\right)I_{add}^{2w+w} + 3\left(\frac{n-1}{2n}\right)I_{shift}^{w}\right),$$

where n – number of w-bit machine words required to store the multiplier of given size, I_{mul}^{w} – a multiplication operation for w-bit words, I_{add}^{2w+w} – an addition operation for $2w$-bit and w-bit words, I_{shift}^{w} – a word shift operation. Assignment operations do not take into account in computational complexity of the algorithms.

The evaluation results of computational complexity of ***MCSQR*** for different bit length multipliers are shown in Table 1 (MUL, ADD and SHIFT – amount of required multiplication, addition and shift operations).

Table 1 shows that using 64-bit machine words in MCSQR algorithm can significantly reduce the number of necessary operations (including reducing the number of multiplications by 4 times).

Table 1. The number of operations for ***MCSQR***

BIT SIZE	MCSQR, *w*=32 bit			MCSQR, *w*=64 bit		
	MUL	**ADD**	**SHIFT**	**MUL**	**ADD**	**SHIFT**
128	16	52	18	4	14	3
256	64	200	84	16	52	18
512	256	784	360	64	200	84
1024	1024	3104	1488	256	784	360
2048	4096	12352	6048	1024	3104	1488
3072	9216	27744	13680	2304	6960	3384
4096	16384	49280	24384	4096	12352	6048
6144	36864	110784	55008	9216	27744	13680
8192	65536	196864	97920	16384	49280	24384
12288	147456	442752	220608	36864	110784	55008
16384	262144	786944	392448	65536	196864	97920

3.2 Squaring algorithms with parallelization techniques

The delayed carry mechanism allows formulating several approaches to the ***MCSQR*** parallelization:

- Parallel execution (in two parallel threads) of loops in the step 2 and 3 with further final result correction.
- Parallel execution (number of parallel threads) of iterations in loops in step 2 and 3 with further intermediate results (from parallel threads) merging.

3.2.1 Algorithm Modified Comba SQR 2x

Algorithm with two parallel processing threads and paralleling features of the ***MCSQR*** algorithm was proposed. ***Modified Comba SQR 2x (MCSQR2x)*** is squaring algorithm for *w*-bit platforms, based on multiplication algorithm ***Modified Comba 2x (MC2x)*** (Kovtun and Okhrimenko, 2012), (Kovtun and Okhrimenko, 2013) with two parallel processing threads:

Squaring algorithm 2. Modified Comba SQR 2x with two parallel processing threads	
INPUT	**OUTPUT**
$a \in \mathrm{GF}(p),\ n = \log_{2^w} a,\ nk = 2n-1$	$c = a^2$
1. #pragma omp parallel sections begin 1.1. #pragma omp section begin 1.1.1. $rl_0^{(2w)} \leftarrow 0,\ rl_1^{(2w)} \leftarrow 0,\ rl_2^{(2w)} \leftarrow 0.$ 1.1.2. For $k \leftarrow 0,\ k < n,\ k{+}{+}$ do 1.1.2.1. For $i \leftarrow 0,\ j \leftarrow k,\ i \le j,\ i{+}{+},\ j{-}{-}$ do	

1.1.2.1.1. $\left(u^2\right)^{(2w)} \leftarrow a_i^{(w)} \cdot a_j^{(w)}$.

1.1.2.1.2. if $(i < j)$ then $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + \left(u^{(w)} << 1\right)$,
$rl_1^{(2w)} \leftarrow rl_1^{(2w)} + \left(\left(u^{(w)} << 1\right) \text{ or } \left(u^{(w)} >> (w-1)\right)\right)$;
else $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + u^{(w)}$, $rl_1^{(2w)} \leftarrow rl_1^{(2w)} + u^{(w)}$

} ***delayed carry accumulation***

1.1.2.2. $rl_1^{(2w)} \leftarrow rl_1^{(2w)} + \text{hi}_{(w)}\left(rl_0^{(2w)}\right)$, $rl_2^{(2w)} \leftarrow rl_2^{(2w)} + \text{hi}_{(w)}\left(rl_1^{(2w)}\right)$ ***// carry accounting***

1.1.2.3. $c_k^{(w)} \leftarrow \text{low}_{(w)}\left(rl_0^{(2w)}\right)$, $rl_0^{(2w)} \leftarrow \text{low}_{(w)}\left(rl_1^{(2w)}\right)$,
$rl_1^{(2w)} \leftarrow \text{low}_{(w)}\left(rl_2^{(2w)}\right)$, $rl_2^{(2w)} \leftarrow 0$.

1.1.3. $r_0^{(2w)} \leftarrow rl_0^{(2w)}$.

#pragma omp section end

1.2. #pragma omp section begin

1.2.1. $rl_0^{(2w)} \leftarrow 0$, $rl_1^{(2w)} \leftarrow 0$, $rl_2^{(2w)} \leftarrow 0$.

1.2.2. For $k \leftarrow n$, $l \leftarrow 1, k < nk$, $k{+}{+}$, $l{+}{+}$ do

1.2.2.1. For $i \leftarrow l$, $j \leftarrow n-1$, $i \le j$, $i{+}{+}$, $j{-}{-}$ do

1.2.2.1.1. $\left(u^2\right)^{(2w)} \leftarrow a_i^{(w)} \cdot a_j^{(w)}$.

1.2.2.1.2. if $(i < j)$ then $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + \left(u^{(w)} << 1\right)$,
$rl_1^{(2w)} \leftarrow rl_1^{(2w)} + \left(\left(u^{(w)} << 1\right) \text{ or } \left(u^{(w)} >> (w-1)\right)\right)$;
else $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + u^{(w)}$, $rl_1^{(2w)} \leftarrow rl_1^{(2w)} + u^{(w)}$

} ***delayed carry accumulation***

1.2.2.2. $rl_1^{(2w)} \leftarrow rl_1^{(2w)} + \text{hi}_{(w)}\left(rl_0^{(2w)}\right)$, $rl_2^{(2w)} \leftarrow rl_2^{(2w)} + \text{hi}_{(w)}\left(rl_1^{(2w)}\right)$ ***// carry accounting***

1.2.2.3. $c_k^{(w)} \leftarrow \text{low}_{(w)}\left(rl_0^{(2w)}\right)$, $rl_0^{(2w)} \leftarrow \text{low}_{(w)}\left(rl_1^{(2w)}\right)$,
$rl_1^{(2w)} \leftarrow \text{low}_{(w)}\left(rl_2^{(2w)}\right)$, $rl_2^{(2w)} \leftarrow 0$.

#pragma omp section end

#pragma omp parallel sections end

2. For $k \leftarrow n$, $k < nk$, $k{+}{+}$ do

2.1. $r_0^{(2w)} \leftarrow r_0^{(2w)} + c_k^{(w)}$.

2.2. $c_k^{(w)} \leftarrow \text{low}_{(w)}\left(r_0^{(2w)}\right)$.

2.3. $\text{low}_{(w)}\left(r_0^{(2w)}\right) \leftarrow \text{hi}_{(w)}\left(r_0^{(2w)}\right)$.

3. $c_{nk}^{(w)} \leftarrow c_{nk}^{(w)} + \text{low}_{(w)}\left(r_0^{(2w)}\right)$.

4. Return (c).

The computational complexity of the ***MCSQR2x*** algorithm:

$$I_{sqr}^{MCSQR2x} = \max\left[n\left(\left\lceil \frac{n}{2} \right\rceil \left(I_{mul}^{w} + 2\left(\frac{n+1}{2n} \right) I_{add}^{2w+w} + 3\left(\frac{n-1}{2n} \right) I_{shift}^{w} \right) + 2I_{add}^{2w+w} \right), \right.$$
$$\left. n\left(\left\lfloor \frac{n}{2} \right\rfloor \left(I_{mul}^{w} + 2\left(\frac{n+1}{2n} \right) I_{add}^{2w+w} + 3\left(\frac{n-1}{2n} \right) I_{shift}^{w} \right) + 2I_{add}^{2w+w} \right) \right] + \tfrac{n}{2} I_{add}^{2w+w} + I_{add}^{w}$$

where n – number of w-bit machine words required to store the multiplier of given size, I_{mul}^{w} – a multiplication operation for w-bit words, I_{add}^{2w+w} – an addition operation for $2w$-bit and w-bit words, I_{shift}^{w} – a shift operation. Assignment operations do not take into account in computational complexity of the algorithms.

The evaluation results of computational complexity of ***MCSQR2x*** for different bit length multipliers are shown in Table 2 (MUL, ADD and SHIFT – amount of required multiplication, addition and shift operations):

Table 2. The number of operations for ***MCSQR2x***

BIT SIZE	**MCSQR2x, *w*=32 bit**			**MCSQR2x, *w*=64 bit**		
	MUL	**ADD**	**SHIFT**	**MUL**	**ADD**	**SHIFT**
128	8	19	9	2	8	2
256	32	53	42	8	19	9
512	128	169	180	32	53	42
1024	512	593	744	128	169	180
2048	2048	2209	3024	512	593	744
3072	4608	4849	6840	1152	1273	1692
4096	8192	8513	12192	2048	2209	3024
6144	18432	18913	27504	4608	4849	6840
8192	32768	33409	48960	8192	8513	12192
12288	73728	74689	110304	18432	18913	27504
16384	131072	132353	196224	32768	33409	48960

3.2.2 Algorithm Modified Comba SQR Mx

Modified Comba SQR Mx (MCSQRMx) is the squaring algorithm for w-bit platforms, based on multiplication algorithm ***Modified Comba Mx (MCMx)*** with multiple parallel processing threads (Kovtun and Okhrimenko, 2012), (Kovtun and Okhrimenko, 2013):

Squaring algorithm 3. Modified Comba SQR Mx with multiple parallel processing threads

INPUT	OUTPUT
$a, b \in \mathrm{GF}(p),\ n = \log_{2^w} a,\ nk = 2n-1$	$c = a \cdot b$

1. $l \leftarrow 1$.

2. Arrays $r0_i^{(w)}$ and $r1_i^{(w)}$, $i = \overline{0, nk}$.

3. #pragma omp parallel begin

4. #pragma omp for nowait begin

4.1. For $k \leftarrow 0$, $k < n$, $k{+}{+}$ do

4.1.1. $rl_0^{(w)} \leftarrow 0$, $rl_1^{(w)} \leftarrow 0$.

4.1.2. For $i \leftarrow 0$, $j \leftarrow k$, $i \le j$, $i{+}{+}$, $j{-}{-}$ do

4.1.2.1. $\left(u^2\right)^{(2w)} \leftarrow a_i^{(w)} \cdot a_j^{(w)}$.

4.1.2.2. if $(i < j)$ then $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + \left(u^{(w)} << 1\right)$,
$rl_1^{(2w)} \leftarrow rl_1^{(2w)} + \left(\left(u^{(w)} << 1\right)\ or\ \left(u^{(w)} >> (w-1)\right)\right)$;
else $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + u^{(w)}$, $rl_1^{(2w)} \leftarrow rl_1^{(2w)} + u^{(w)}$
— ***delayed carry accumulation***

4.1.3. $r0_k^{(2w)} \leftarrow rl_0^{(2w)}$, $r1_k^{(2w)} \leftarrow rl_1^{(2w)}$ — ***// saving carry***

#pragma omp for end

5. #pragma omp for nowait begin

5.1. For $k \leftarrow n$, $k < nk$, $k{+}{+}$ do

5.1.1. $rl_0^{(2w)} \leftarrow 0$, $rl_1^{(2w)} \leftarrow 0$.

5.1.2. For $i \leftarrow l$, $j \leftarrow n-1$, $i \le j$, $i{+}{+}$, $j{-}{-}$ do

5.1.2.1. $\left(u^2\right)^{(2w)} \leftarrow a_i^{(w)} \cdot a_j^{(w)}$.

5.1.2.2. if $(i < j)$ then $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + \left(u^{(w)} << 1\right)$,
$rl_1^{(2w)} \leftarrow rl_1^{(2w)} + \left(\left(u^{(w)} << 1\right)\ or\ \left(u^{(w)} >> (w-1)\right)\right)$;
else $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + u^{(w)}$, $rl_1^{(2w)} \leftarrow rl_1^{(2w)} + u^{(w)}$
— ***delayed carry accumulation***

5.1.3. $r0_k^{(2w)} \leftarrow rl_0^{(2w)}$, $r1_k^{(2w)} \leftarrow rl_1^{(2w)}$ — ***// saving carry***

5.1.4. $l++$.

#pragma omp for end

#pragma omp parallel end

6. $r^{(2w)} \leftarrow 0$

7. For $k \leftarrow 0$, $k < nk$, $k++$ do *// **delayed carry accounting***

7.1. $rl_0^{(2w)} \leftarrow r0_k^{(2w)}$.

7.2. $rl_0^{(2w)} \leftarrow rl_0^{(2w)} + r^{(2w)}$.

7.3. $c_k^{(w)} \leftarrow low_{(w)}\left(rl_0^{(2w)}\right)$.

7.4. $r^{(2w)} \leftarrow r1_k^{(2w)} + hi_{(w)}\left(rl_0^{(2w)}\right)$.

8. $c_{nk}^{(w)} \leftarrow \mathrm{low}_{(w)}\left(r^{(2w)}\right)$.

9. Return (c).

The computational complexity of the ***MCSQRMx*** algorithm:

$$I_{mul}^{MCSQRMx} = \frac{n}{Z}\left[\left\lceil\frac{n}{2}\right\rceil\left(I_{mul}^{w} + 2\left(\frac{n+1}{2n}\right)I_{add}^{2w+w} + 3\left(\frac{n-1}{2n}\right)I_{shift}^{w}\right)\right] +$$

$$+\frac{n}{Z}\left[\left\lfloor\frac{n}{2}\right\rfloor\left(I_{mul}^{w} + 2\left(\frac{n+1}{2n}\right)I_{add}^{2w+w} + 3\left(\frac{n-1}{2n}\right)I_{shift}^{w}\right)\right] + (2n-1)3I_{add}^{2w+w},$$

where Z – parallel threads count, n – number of w-bit machine words required to store the multiplier of given size, I_{mul}^{w} – a multiplication operation for w-bit words, I_{add}^{2w+w} – an addition operation for $2w$-bit and w-bit words, I_{shift}^{w} – a word shift operation. Assignment operations do not take into account in computational complexity of the algorithms.

The evaluation results of computational complexity of ***MCSQR2x*** for different bit length multipliers are shown in Table 3 for Z=4 (MUL, ADD and SHIFT – amount of required multiplication, addition and shift operations):

Table 3. The number of operations for ***MCSQRMx***

BIT SIZE	**MCSQRMx, *w*=32 bit**			**MCSQRMx, *w*=64 bit**		
	MUL	**ADD**	**SHIFT**	**MUL**	**ADD**	**SHIFT**
128	4	26	5	1	11	1
256	16	63	21	4	26	5
512	64	161	90	16	63	21
1024	256	453	372	64	161	90
2048	1024	1421	1512	256	453	372
3072	2304	2901	3420	576	873	846

4096	4096	4893	6096	1024	1421	1512
6144	9216	10413	13752	2304	2901	3420
8192	16384	17981	24480	4096	4893	6096
12288	36864	39261	55152	9216	10413	13752
16384	65536	68733	98112	16384	17981	24480

Theoretical calculations show that the parallel squaring algorithms have a lower computational complexity, primarily due to the parallel execution of the elementary operations of addition and multiplication. Furthermore, the use of 64-bit machine words reduces the number of multiplications by 4 times.

4 FIELD RESEARCH

Squaring algorithms ***MCSQR, MCSQR2x*** and ***MCSQRMx*** as previously proposed algorithms for multiplication ***MC, MC2x*** and ***MCMx*** (Kovtun et al., 2012), (Kovtun and Okhrimenko, 2012), (Kovtun and Okhrimenko, 2013) have been implemented in software in *C++* using the *Intel C + + Compiler XE 13*. The proposed algorithms have been implemented for 32- and 64-bit platforms. Measurements were performed on a computer running *Microsoft Windows 7 Ultimate x64 SP1* and the processor *Intel Core i5-3570 (6M Cache, 3.40 GHz)* with four physical cores. For multiplication of two 64-bit integers, have been used the built-in compiler intrinsic function *_umul128*, (128-bit result of the multiplication is represented as an array of 64-bit words). Comparison of the results occurred by comparing the average time of multiplication operations in software implementation ***MC, MC2x*** and ***MCMx*** and the proposed algorithms squaring ***MCSQR, MCSQR2x*** and ***MCSQRMx***, for 1 million iterations.

The experimental results for 32-bit platforms are shown in Table 4.

Table 4. The experimental results for *w=32 bit*

BIT SIZE	MCSQR, ms	MCSQR2x, ms	MCSQRMx, ms	MC, ms	MC2x, ms	MCMx, ms
128	59	681	726	62	723	768
256	147	702	750	156	749	796
512	495	858	821	530	936	874
1024	1722	1576	936	1919	2028	999
2048	6490	4056	1432	7394	5772	1513
3072	14305	8143	2196	16349	12028	2340
4096	24992	13688	3023	28673	20560	3245
6144	54928	29593	5519	63133	44569	5938
8192	97266	51542	7438	110979	78842	7878
12288	214921	111930	13960	246730	174174	14708
16384	378488	196768	21359	435943	306416	23306

It is proposed to normalize the results by dividing the results of ***MC, MC2x*** and ***MCMx*** to results ***MCSQR, MCSQR2x*** and ***MCSQRMx***. The normalized results are shown in Table 5.

Table 5. Normalized results of experiments for *w = 32 bit*

BIT SIZE	MC/MCSQR	MC2x/MCSQR2x	MCMx/MCSQRMx
128	1,051	1,062	1,058
256	1,061	1,067	1,061
512	1,071	1,091	1,065
1024	1,114	1,287	1,067
2048	1,139	1,423	1,057
3072	1,143	1,477	1,066
4096	1,147	1,502	1,073
6144	1,149	1,506	1,076
8192	1,141	1,530	1,059
12288	1,148	1,556	1,054
16384	1,152	1,557	1,091

Table 5 shows that all proposed squaring algorithms for 32-bit platforms are effectively than multiplying algorithms. Single-threaded algorithm ***MCSQR*** is more efficient than algorithm ***MC*** by 5%, and advantage increases to 15% with the increase of the multipliers bit size. The algorithm ***MCSQR2x*** is more efficient than algorithm ***MC2x*** by 6%, and advantage increases to 56% with the increase of the multipliers bit size. Multi-threaded algorithm ***MCSQRMx*** is more effectively than algorithm ***MCMx*** by 6%, and advantage increases to 9% with the increase of the multipliers bit size.

The experimental results for 64-bit platforms are shown in Table 6.

Table 6. The experimental results for *w=64 bit*

BIT SIZE	MCSQR, ms	MCSQR2x, ms	MCSQRMx, ms	MC, ms	MC2x, ms	MCMx, ms
128	14	628	687	15	818	903
256	46	687	721	50	896	949
512	150	699	780	163	924	1070
1024	483	889	952	593	1184	1293
2048	1736	1541	1560	2347	2053	2224
3072	3776	2816	2690	5175	3801	3810
4096	6521	4112	4009	8830	5600	5507
6144	14521	8549	8205	19532	11590	11357
8192	25490	14430	13759	34335	19858	18752
12288	56379	31794	29718	76674	44991	40435
16384	99232	54507	51651	135252	77127	71682

It is proposed to normalize the results for 64-bit platforms by dividing the results of ***MC, MC2x*** and ***MCMx*** to results ***MCSQR, MCSQR2x*** and ***MCSQRMx*** too. The normalized results are shown in Table 7.

Table 7. Normalized results of experiments for *w = 64 bit*

BIT SIZE	MC/MCSQR	MC2x/MCSQR2x	MCMx/MCSQRMx
128	1,071	1,303	1,314
256	1,087	1,304	1,316
512	1,087	1,322	1,372
1024	1,228	1,332	1,358
2048	1,352	1,332	1,426
3072	1,370	1,350	1,416
4096	1,354	1,362	1,374
6144	1,345	1,356	1,384
8192	1,347	1,376	1,363
12288	1,360	1,415	1,361
16384	1,363	1,415	1,388

Table 7 shows that all proposed squaring algorithms for 64-bit platforms are effectively than multiplying algorithms. Single-threaded algorithm ***MCSQR*** is more efficient than algorithm ***MC*** by 7%, and advantage increases to 36% with the increase of the multipliers bit size. The algorithm ***MCSQR2x*** is more efficient than algorithm ***MC2x*** by 30%, and advantage increases to 41% with the increase of the multipliers bit size. Multi-threaded algorithm ***MCSQRMx*** is more effectively than algorithm ***MCMx*** in average of 37%.

To compare the performance of software implementations of squaring algorithms for 32 and 64 bit platforms, the results obtained on 32-bit platform were divided by results on 64-bit platform, for the same algorithms. The comparison results are shown in Table 8.

Table 8. Performance comparing of software implementations

BIT SIZE	MCSQRx86/ MCSQRx64	MCSQR2xx86/ MCSQR2xx64	MCSQRMxx86/ MCSQRMxx64	MCx86/ MCx64	MC2Xx86/ MC2Xx64	MCMXx86/ MCMXx64
128	4,214	1,084	1,057	4,133	0,884	0,850
256	3,196	1,022	1,040	3,120	0,836	0,839
512	3,300	1,227	1,053	3,252	1,013	0,817
1024	3,565	1,773	0,983	3,236	1,713	0,773
2048	3,738	2,632	0,918	3,150	2,811	0,680
3072	3,788	2,892	0,816	3,159	3,164	0,614
4096	3,833	3,329	0,754	3,247	3,671	0,589
6144	3,783	3,462	0,673	3,232	3,845	0,523
8192	3,816	3,572	0,541	3,232	3,970	0,420
12288	3,812	3,520	0,470	3,218	3,871	0,364
16384	3,814	3,610	0,414	3,223	3,973	0,325

Table 8 shows that, as in case of multiplication, software implementations of squaring algorithms for 64-bit platforms were more effective than the same implementation for 32-bit

platforms (up to 4 times for the algorithm ***MCSQR***, up to 3,6 times for the algorithm ***MCSQR2x***). Multithreaded multiplication and squaring algorithms for 32-bit platforms was more effective than for 64-bit, because there are not support 128-bit operations in modern compilers, that's why it require software emulation such operations. ***MCSQRMx*** algorithm for 64-bit platforms is more effectively on 128-512 bits multipliers (4-6%), which are widely used in cryptography.

Theoretical estimation for ***MCSQR*** and ***MCSQR2x*** confirmed by practical results of research. MCSQRMx algorithm test results for 64-bit platforms indicate that there are no operations performed on 128-bit data array.

5 CONCLUSIONS

Using previously proposed approaches to improve the performance of the integers multiplication (Kovtun et al., 2012), (Kovtun and Okhrimenko, 2012), (Kovtun and Okhrimenko, 2013) have developed squaring algorithms ***Modified Comba SQR, Modified Comba SQR 2x*** and ***Modified Comba SQR Mx***.

Theoretical calculations show that the parallel squaring algorithms have a lower computational complexity, primarily due to the parallel execution of the elementary operations of addition and multiplication. Furthermore, the use of 64-bit machine words reduces the number of multiplications by 4 times.

Software implementations of squaring algorithms for 64-bit platforms were more effective than the same implementation for 32-bit platforms (up to 4 times for the algorithm ***MCSQR***, up to 3,6 times for the algorithm ***MCSQR2x***). ***MCSQRMx*** algorithm for 64-bit platforms is more effectively on 128-512 bits multipliers (4-6%), which are widely used in cryptography. Multithreaded multiplication and squaring algorithms for 32-bit platforms was more effective than for 64-bit, because there are not support 128-bit operations in modern compilers, that's why it require software emulation such operations.

Experimental researches have shown the effectiveness of the proposed squaring algorithms over multiplication algorithms for 32-bit and 64-bit platforms. The theoretical results are confirmed by practice.

The most perspective algorithm is ***MCSQRMx***, which shows significantly better results than other presented algorithms. ***MCSQRMx*** has a high degree of parallelism, which allows implementing it on various microprocessor platforms, that's why further research will focus on its development using specialized software and hardware (e.g., *NVIDIA CUDA* and *OpenCL*).

REFERENCES

1. Denis, T., Rose G. (2006). *BigNum Math: Implementing Cryptographic Multiple Precision Arithmetic*. Elsevier/Syngress.
2. Handbook of Elliptic and Hyperelliptic Curve Cryptography. (2006). Chapman & Hall/CRC.
3. Hankerson, D., Menezes, A.J., Vanstone, S. (2004). *Guide to Elliptic Curve Cryptography*. Springer-Verlag Professional Computing Series.
4. Kovtun, V.Y., Okhrimenko, A.O. (2012). Approaches for the Parallelization of Software Implementation of Integer Multiplication. *Radiotehnika. Vseukrainskij mezhvedomstvennyj nauchno-tehnicheskij sbornik, 171*, 123-132.
5. Kovtun, V.Y., Okhrimenko, A.O. (2013). Integer multiplication algorithms with delayed carry for public-key cryptosystems. In: V.S. Ponomarenko (Eds.), Informacionnye tehnologi i sistemy v upravlenii, obrazovanii, nauke (pp. 69-82). Har'kov: Cifrova drukarnja №1.
6. Kovtun, V.Y., Okhrimenko, A.O., Nechiporuk , V.V. (2012). Approaches for the performance increasing of software implementation of integer multiplication in prime fields. *Zashhita informacii, 1 (54),* 68-75.