Fast Software Implementation of Binary Elliptic Curve Cryptography

Manuel Bluhm[1] and Shay Gueron[2,3]

[1] Ruhr University Bochum, Germany
[2] University of Haifa, Israel
[3] Intel Corporation, Israel Development Center, Israel

November 10, 2013

Abstract. This paper presents an efficient and side channel protected software implementation of point multiplication for the standard NIST and SECG binary elliptic curves. The enhanced performance is achieved by improving the Lòpez-Dahab/Montgomery method at the algorithmic level, and by leveraging Intel's AVX architecture and the pclmulqdq processor instruction at the coding level. The fast carry-less multiplication is further used to speed up the reduction on the newest Haswell platforms. For the five NIST curves over $GF(2^m)$ with $m \in \{163, 233, 283, 409, 571\}$, the resulting point multiplication implementation is about 6 to 12 times faster than that of OpenSSL-1.0.1e, enhancing the ECDHE and ECDSA algorithms significantly.

Keywords: ECC, binary elliptic curves, point multiplication, pclmulqdq, AVX, ECDH, ECDSA, side channel protection, NIST, Lòpez-Dahab.

1 Introduction

All SSL/TLS communications start with a client-server key establishment, based on a public key cryptosystem. The classical protocol uses the RSA algorithm for encryption/decryption of a session key, but it suffers from the following drawback: the confidentiality of the sessions depends directly on the security of the server's private RSA key. Especially motivated by recent disclosures, the current trends are to move to protocols providing *(perfect) forward secrecy*, such as the authenticated Diffie-Hellman Ephemeral (DHE) key exchange. One approach is to use Elliptic Curve Cryptography (ECC) to efficiently perform this kind of key exchange (ECDHE). Similarly, ECC based signature schemes (ECDSA) become attractive because of their relatively short keys. Thus, software implementations of ECC for the high end server platforms become a target for optimization.

Protocols such as TLS 1.x support elliptic curves over both prime and binary fields, and the important cryptographic libraries include implementations for both types of ECC. For instance, the OpenSSL library that underlies the `mod_ssl` module of Apache, implements the ECDHE-ECDSA, ECDHE-RSA and other key agreement protocols over binary and prime fields. It is interesting to note that from the hardware implementation cost, protocols using binary curves are cheaper to implement than prime field curves, which can be significant for low-end client devices. However, on the software implementation side, binary curves have been less efficient than their prime curves counterparts on high end processors. This was mainly caused by the fast integer multiplication units compared to slow carry-less multiplication. In OpenSSL-1.0.1e for example, the prime curve `secp256k1` outperformes it's binary counterpart `sect283k1` by several magnitudes.

This paper focuses on software optimization of point multiplication over `binary` elliptic curves, leveraging the AVX instruction set and the newly optimized carry-less multipliers on high end x86-64 platforms. Targeting performance and security by fast, side channel protected code, we built a library for the underlying binary field arithmetic for the common fields specified by NIST and SECG [17], covering security levels from 80 bits up to 256 bits. Table 1 shows the binary elliptic curves over $GF(2^m)$ (both random and Koblitz curves) that are optimized in this paper. It further displays their field representation and the equivalent bit level security, and (in parentheses) the corresponding RSA key length. On the left column, "**1**" indicates that the respective curve is standardized by NIST (the National Institute of Standards and Technology); "**2**" indicates a curve standardized by SECG (Standards for Efficient Cryptography Group); "**3**" indicates that the curve is part of the Wireless Transport Layer Security (WTLS).

Curve identifier	**Type**	**m**	**Bit level security** (Equiv. RSA key length)	**Field polynomial** for reduction in $GF(2^m)$
`sect163k1`^{1,2,3}	Koblitz	163	80 (1024)	$x^{163}+x^7+x^6+x^3+1$
`sect163r1`^{2}	Random			
`sect163r2`^{1,2}	Random			
`sect193r1`^{2}	Random	193	96 (1536)	$x^{193}+x^{15}+1$
`sect193r2`^{2}	Random			
`sect233k1`^{1,2,3}	Koblitz	233	112 (2240)	$x^{233}+x^{74}+1$
`sect233r1`^{1,2,3}	Random			
`sect239k1`^{2}	Koblitz	239	115 (2304)	$x^{239}+x^{158}+1$
`sect283k1`^{1,2}	Koblitz	283	128 (3456)	$x^{283}+x^{12}+x^7+x^5+1$
`sect283k1`^{1,2}	Random			
`sect409k1`^{1,2}	Koblitz	409	192 (7680)	$x^{409}+x^{87}+1$
`sect409r1`^{1,2}	Random			
`sect571k1`^{1,2}	Koblitz	571	256 (15360)	$x^{571}+x^{10}+x^5+x^2+1$
`sect571r1`^{1,2}	Random			

Table 1: Targeted binary elliptic curves and their security level

2 The improved 2P algorithm

Given an elliptic curve $\mathcal{E}$ with $|\mathcal{E}| = |G_{\mathcal{E}}| \cdot |H_{\mathcal{E}}| = n \cdot h$ (here: $h \in \{2,4\}$), the essential operation of elliptic curve cryptosystems is multiplying a point $P \in G_{\mathcal{E}}$ on the curve, by a positive integer $k \in [1,n]$, to obtain a new point

$$Q = k \cdot P = \underbrace{P + P + ... + P}_{\text{k-times}} = \sum_{i=0}^{k-1} P.$$

Montgomery [15] introduced a fast approach for this point multiplication. It uses the fact that the sum of two points whose difference is known, can be evaluated with a procedure that involves only the x-coordinate of these points when performing one point doubling, and one point addition per iteration. López and Dahab (LD hereafter) improved this algorithm for binary curves over $GF(2^m)$, by eliminating costly inversion operations in the field (when given in polynomial representation). In their so-called **2P-Algorithm** [14], the point $P = (x,y)$ is represented in the LD-projective form $P = (X/Z, Y/Z^2)$. The x-coordinate of the sum $P_0 + P_1$ and of $2P_i$ is the fraction X'/Z', where

$$\begin{array}{ll} \underline{2P_i}: & \underline{P_0 + P_1}: \\ & \\ X' = X_i^4 + b \cdot Z_i^4 & X' = x \cdot Z' + (Z_0 \cdot X_1) \cdot (Z_1 \cdot X_0) \\ Z' = X_i^2 \cdot Z_i^2 & Z' = \Big((Z_0 \cdot X_1) + (Z_1 \cdot X_0) \Big) \end{array}$$

The computational cost of the resulting algorithm is dominated by the performance of the point addition(`Madd`) and point doubling(`Mdouble`), which are executed $\lfloor log_2(k) \rfloor$ times in a point multiplication kP. Here, we speed up the point multiplication by reducing the amount of computations in `Madd` and `Mdouble`.

2.1 Improved Mdouble

By re-ordering the flow of the `Mdouble`, we first multiply Z^2 with the precomputed value $c = \sqrt{b}$, then add the result to X^2, square this again and obtain the result $X \leftarrow (X^2 + c \cdot Z^2)^2 = X^4 + b \cdot Z^4$. This saves one squaring and one reduction operation per run. Algorithm 2 shows the improved `Mdouble` flow.

For Koblitz curves, we have $b = 1$ and therefore $c = b^{2^{m-1}} = 1$, so another multiplication and reduction can be saved, as shown in Algorithm 3.

2.2 Improved Madd

The `Madd` function can be improved by deferring reduction operations (we call it *lazy reduction*, after [21]). Algorithm 4 shows the improved `Madd` flow, where we defer the reduction after Steps 4 and 8, and reduce the result only in Step 9. The rationale is that double sized addition is cheaper than a reduction (the actual gain depends on the specific field).

2.3 Estimating the impact of the improved Mdouble and Madd

Table 2 compares the number of the field operations in Algorithms 2, 3, and 4, compared to their original form. For simplicity, we count the reduction as a field operation here.

Field Operations	**Mdouble** Koblitz	**Mdouble** Random	**Mdouble** Original	**Madd** Improved	**Madd** Original
#MUL	$\mathbf{1}\lfloor log_2(k)\rfloor$	$2\lfloor log_2(k)\rfloor$	$2\lfloor log_2(k)\rfloor$	$4\lfloor log_2(k)\rfloor$	$4\lfloor log_2(k)\rfloor$
#RED	$\mathbf{4}\lfloor log_2(k)\rfloor$	$\mathbf{5}\lfloor log_2(k)\rfloor$	$6\lfloor log_2(k)\rfloor$	$\mathbf{4}\lfloor log_2(k)\rfloor$	$5\lfloor log_2(k)\rfloor$
#SQR	$\mathbf{3}\lfloor log_2(k)\rfloor$	$\mathbf{3}\lfloor log_2(k)\rfloor$	$4\lfloor log_2(k)\rfloor$	$\lfloor log_2(k)\rfloor$	$\lfloor log_2(k)\rfloor$
#ADD	$\lfloor log_2(k)\rfloor$	$\lfloor log_2(k)\rfloor$	$\lfloor log_2(k)\rfloor$	$\mathbf{3}\lfloor log_2(k)\rfloor$	$2\lfloor log_2(k)\rfloor$

Table 2: The number of field operations for point multiplication, with the original and the improved `Mdouble` and `Madd` algorithms.

3 Side channel protected point multiplication

This section discusses the potential threat and countermeasures of software side channel attacks, where an adversary attempts to collect information about the secret keys that are being used. Today, it is known that software implementations of cryptographic primitives need to take such side channel attacks into account, and add appropriate countermeasures.

3.1 Protection requirements

As with other cryptographic schemes, implementations of ECC have been shown to be susceptible to software side channel attacks. Even the applicability of remote timing attacks on a ECC implementation has been demonstrated [7]. We concentrate here on protecting the software implementation of the LD/Montgomery point multiplication, without covering the higher implementation parts of the various protocols like ECDHE or ECDSA.
Timing attacks exploit the occurrence of variant execution times for different inputs. This kind of attack can be mounted whenever a correlation between the key and the execution time is found. Further, we take cache based timing attacks into account, where the adversary is assumed to be able to run an unprivileged process to measure instruction and data cache latencies. Thus, key dependent branches and table accesses should be avoided or masked.

To avoid timing attacks in the LD/Montgomery point multiplication one must ensure that

1. all inputs (k,P) to the point multiplication function are valid, meaning
   - Integer k is in the interval $[1, n]$ with $n = |\langle P\rangle|$
   - k has constant bit length (for the same group)
   - P is a valid point on the curve.
2. the point multiplication and its sub-functions are executed in constant time,
3. no key dependent code branches or data accesses exist.

3.2 Protection strategy

The large variety of attacks makes it very difficult to protect an implementation against all possible side channel threats. In the following, we will show the mechanisms to prevent the previously mentioned threats for server implementations.

Constant key length is a first step to assert a constant running time of the point multiplication kP. Although, the input validation of the scalar k and the point P should be performed prior to the point multiplication, we provide a bit length fix for all *valid* $k \in [1, n]$ with n order of the large subgroup $G_{\mathcal{E}}$ of $\mathcal{E}$. At first, we always add n to k, so that $k' = k + n$ and $kP = k'P$. The bit length of k' is then $\lfloor log_2(n) \rfloor \leq log_2(k') \leq \lfloor log_2(n) \rfloor + 1$, since the result of an addition of two integers n and $k \leq n$ is not longer than $\lfloor log_2(n) \rfloor + 1$ bits.

To make the bit length fixed for all cases of k, one can repeat the addition of n, if and only if the bit at position $\lfloor log_2(n) \rfloor + 1$ is NOT set. To ensure that the implementation does not include any branches, one should create a mask in a similar way as suggested in Algorithm 1 and then add the masked value, which is either the order n or an equivalent sized number with zero value. Thereafter, the bit length of k' is fixed to $\lceil log_2(n) \rceil$. One has to take into account that k needs to be a valid input with $k \in [1, n]$ in order to assert the fixed bit length, which is expected to be evaluated outside the point multiplication. Additionally, this countermeasure eventually adds vulnerability to Carry-based attacks [8], attacking not the point multiplication but this particular countermeasure itself. However, considering our attacker model this is no reasonable threat.

Constant memory access pattern and the **elimination of key dependent branches** are the main countermeasure against cache based side channel attacks in our implementation. Since cache attacks are most likely with our attacker model, we want to assure not only a fixed memory access pattern but also avoid any key dependent branch. Therefore, we suggest algorithm 1 to veil the data access.

Let tx_1,tx_2,tz_1 and tz_2 be fixed size temporary variables for a $GF(2^m)$ and k_i the current bit at position i. Table 3 shows the transition of the coordinates involved in the double/add process inside the LD/Montgomery point multiplication's key evaluation loop.

For the proposed data veiling method, a masking word t_0 is created with either all bits set to zero, or all bits set to one, depending on the value of k_i. This word is then used to create the *mask* for field elements in step 2, which in turn is successively applied with AND and NAND operations to the two possible values, satisfying the rules in Table 3. This transition is self-inverse and needs to be re-applied after the execution of `Madd` and `Mdouble`.

	$k_i = 0$	$k_i = 1$
tx_1	x_2	x_1
tx_2	x_1	x_2
tz_1	z_2	z_1
tz_2	z_1	z_2

Table 3: X/Z Transformation

Algorithm 1: Proposed data veiling method

```
Input: Keybit $k_i$, EC coordinates $x_1,\ x_2,\ z_1,\ z_2 \in GF(2^m)$
Output: Temporary coordinates $tx_1,\ tx_2,\ tz_1,\ tz_2 \in GF(2^m)$
1 $t_0 \leftarrow (\underbrace{0x00...0}_{W\text{bit word}} - k_i)$;
2 for $(j = 0;\ j < \lceil \frac{m}{W} \rceil;\ j{+}{+})$ do $mask[j] \leftarrow t_0$;
3 $tx_1 \leftarrow (mask \wedge x_1) \oplus (mask \barwedge x_2)$;
4 $tx_2 \leftarrow (mask \barwedge x_1) \oplus (mask \wedge x_2)$;
5 $tz_1 \leftarrow (mask \wedge z_1) \oplus (mask \barwedge z_2)$;
6 $tz_2 \leftarrow (mask \barwedge z_1) \oplus (mask \wedge z_2)$;
7 return $tx_1,\ tx_2,\ tz_1,\ tz_2$
```

In summary, the proposed technique requires 4 field XORs, NANDs and ANDs, and, neglectable in proportion, one 64 bit subtraction and one constant load each.

Constant function calls inside the key evaluation loop are naturally provided by the LD/Montgomery point multiplication. In addition, we must ensure that all called subfunctions are executed in constant time and thus not include data depending branches. However, due to the branches inside `Mxy`, which is the final transformation to extract the y-coordinate, the running time for the point multiplication is *not constant* if the result is invalid, the inverse $-P$ of a point P or the point at infinity (and thus the scalar k a multiple of n).

Constant time implementation for binary field arithmetic is the necessary consequence from the previous point. If we want to ensure that the running time is constant, the binary field arithmetic must be totally data independent. This does not only mean that we cannot include any shortcuts for special values and data dependent branches at all, but it also biases the choice of algorithm, e.g., the inversion method.

4 Efficient implementation of binary field arithmetics

In this section we explain the techniques for the implementation of the binary field arithmetic. With special focus on fast reduction, we introduce a reduction scheme using the `pclmulqdq` instruction to speed up the reduction on the new Haswell architecture.

4.1 Binary field arithmetic

The square of a field element $a(x) \in GF(2^m)$, rather than just multiplying a field element with itself, can be computed as follows:

$$a^2(x) = \left(\sum_{i=0}^{m-1} a_i\, x^i \right)^2 = \sum_{i=0}^{m-1} (a_i\, x^i)^2 = \sum_{i=0}^{m-1} a_i\, x^{2i} \bmod f(x)$$

This property makes squaring in $GF(2^m)$ a linear operation and therefore much faster than the conventional multiplication, and can be performed with small table lookups. This table can be stored in only one vector and is therefore naturally accessed in a fixed pattern. Expanding the element by successively inserting zeros, it can even be computed in parallel, using the `pshufb` instruction. The use of `pshfub` to implement lookup tables was originally proposed by Gueron and Kounavis [9] and first used by Aranha et al. in [4] for squaring in $GF(2^m)$.

The multiplication in $GF(2^m)$ was usually a very expensive operation. The 64 bit carry-less multiplier, which is now available on various x86 platforms, significantly accelerates the multiplication in binary fields. In this work we use both explicit and recursive versions of Karatsuba-Ofman for different sizes. We chose explicit forms for 2-, 3- and 5-term Karatsuba and recursive forms for 4-,7- and 9-terms, balancing performance gain against code size. For a comparison of Karatsuba implementations with vector extensions see [18] by Su and Fan.

For the inversion in $GF(2^m)$ we use the Itoh-Tsujii Algorithm(ITA)[11]. The ITA performs the inversion with exactly $m-1$ squares and $\lfloor log_2(m-1) \rfloor + H(m-1) - 1$ multiplications with Hamming weight H. The general recursive formula to obtain the chain can be given as

$$1 + 2^n + 2^{2n} + ... + 2^{(k-2)n} = \\ \begin{cases} (1+2^n) \times (1 + 2^{2n} + ... + 2^{(k-3)n}), & \text{if } k-1 \equiv 0 \mod 2 \\ 1 + \big(2^n \times (1+2^n) \times (1 + 2^{2n} + ... + 2^{(k-4)n}\big), & \text{if } k-1 \equiv 1 \mod 2 \end{cases}$$

The running time boundaries for this algorithm are mainly determined by squares and requires a serious amount of time. In this implementation, fortunately, the costly inversion is only required once per point multiplication.

4.2 Reduction

The reduction is a very crucial part of the implementation. The two most important field operations, squaring and multiplication, both require a reduction with the field polynomial $f(x)$, which can be seen as a multiplication with the reduction polynomial. In this work we deal with the NIST polynomials and two polynomials from the SECG standard[17], either trinomials or pentanomials. In this section, we will describe our strategy for each type and discuss special opportunities to speed up the reduction schemes with AVX.

The reduction is basically a multiplication of the upper half of the double sized element with the reduction polynomial. Let $f(x) = x^m + r(x)$ with degree $deg(r) = k$ be the reduction polynomial over $GF(2^m)$. We split the polynomial $c(x) = c_H(x) + c_L(x)$ with degree $deg(c) = m + t$ into two parts where

$$c_H(x) = \sum_{i=m}^{t} c_i x^i \qquad \text{and} \qquad c_L(x) = \sum_{i=0}^{m-1} c_i x^i.$$

The reduction of $c(x)$ is then computed as

$$c'(x) = c_L(x) \oplus c_H(x) \cdot r(x).$$

The resulting polynomial $c'(x)$ has degree $d = deg(c') = max(m-1, t+k)$. In case of $d \geq m$, the reduction is applied recursively.

"Left-to-Right In-Place Shift and Add" approach is the consequence. Since the multiplication of a word by one bit is a simple shift, this multiplication can be applied as a series of shifts and adds. This makes especially sense, if we remember that the amount of bits in $r(x)$ is very small (here either 2 or 4). In our experiments, this technique produced better results than other reduction schemes such as Barrett's method [12][13].

The reduction for **trinomials** is very straightforward - we successively shift the top word by the components of $r(x)$ to the right and add the result to the remainder, while skipping multiples of 64 and 128 bit with memory alignments. In order to avoid duplication of work, one should execute the shift and add from the left to the right and adding the current word immediately to the remainder. Under the assumption that we are about to reduce a double quadword w by component x^k, we shift w exactly $(m - k \bmod 64)$ bits to the right and add it to the remainder while skipping $l \times 128$ bits, where $l = \lfloor \frac{m-k}{128} \rfloor$. If $(m-k \bmod 128) \geq 64$, however, we need to add the lower half of w to the previous word.

The strategy for **pentanomials** differs from the trinomial way in terms of recombination. Instead of directly adding each component to the remainder, we gather the results of left and right shifts of a vector `A` in separated vectors and recombine them by finally adding them to the remainder, which saves three alignments per 128 bit element. Figure 1 shows the combination steps for a vector `A`.

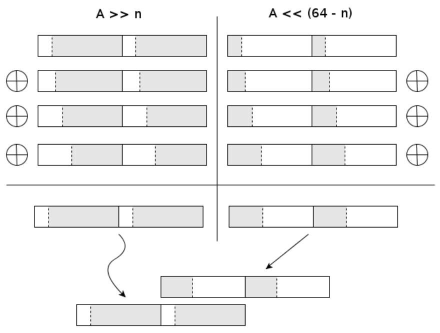

Fig. 1: Example step of the re-combination strategy for pentanomials

Fast vector shifting uses the (faster) 128 bit shift instruction(`pslldq`/`psrldq`), which is feasible whenever the difference of two components in the polynomial is a multiple of 8. With the polynomials from 1, we can apply this trick for the polynomial $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$ where $163 - 3 = 160$. Additionally, we have the components x^{10} and x^2 in the reduction polynomial for $GF(2^{571})$ but here the fast shifting interferes with the recombination strategy, where the savings for the alignment pay off in comparison.

Special reduction schemes are possible in $GF(2^{283})$ and necessary in $GF(2^{239})$. A fast reduction in $GF(2^{283})$ was proposed in [1] and uses the observation $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1 = x^{283} + (x^7 + 1) \cdot (x^5 + 1)$.

The polynomial $f(x) = x^{239} + x^{158} + 1$ is one of the two polynomials suggested in [17] for this field size and currently integrated in OpenSSL. For the implementation with vector extensions, this polynomial is suboptimal. This is caused by the small distance $239 - 158 = 81 < 128$, which means that a 128 bit word of the remainder affects itself during the reduction, which makes shift and add reduction inefficient for this size. The $GF(2^{239})$ reduction is therefore provided in 64 bit mode only. When using vector extensions, the polynomial $f(x) = x^{239} + x^{36} + 1$ (which has also been suggested by SECG) is much faster. Another approach would be to implement the reduction with a special version of Barrett's method, which should increase the performance of this field significantly. Due to this issues we do not recommend the use of this field. Reduction in $GF(2^{233})$ is much faster and this particular curve has almost the same security level (see 1) with an only slightly shorter key. Anyhow, we added this curve to our implementation for convenience.

4.3 Mul&Add reduction

The `pclmulqdq` instruction can be utilized for reduction. Thus, each 64 bit word of the operand $c_H(x)$ must be multiplied with $r(x) = \sum_{i=0}^{k} r_i x^i$ and added to the remainder $c_L(x)$. Obviously, k must hold the condition $k < 64$ to ensure the lowest number of multiplications, which is exactly $t = \lceil \frac{log_2(c_H(x))}{64} \rceil$ then. If $k > 63$, we must multiply each word more than once, particularly $s = \lceil \frac{k}{64} \rceil$ times per word and we therefore require $t \cdot s$ multiplications overall.

In order to spare the additional reduction step, as described above, the multiplications are executed from the left to the right and the results are immediately added to the remainder. Due to the fact that the bits of $c(x)$ are results from either multiplication or squaring and are stored into a series of words where the border between $c_L(x)$ and $c_H(x)$ at position $m - 1$ is right in the middle of a 64 bit block, each multiplication result must be aligned in order to be added to the remainder. Therefore, we can either initially shift the polynomial $r(x)$ to the left by $l = m \bmod 64$ bits (if $k + l < 64$ still holds), or we can shift the input words l bits to the right, or $64 - l$ bits to the left. Thereby, the operand needs to be saved temporarily as the reduction is performed in-place, requiring more temporary variables. Shifting the operand especially makes sense if the data in

$c_H(x)$ overlaps into an additional register, thus requiring $t+1$ multiplications. For example the reduction in $GF(2^{163})$, the bits of $c_H(x) = \sum_{i=163}^{325} c_i x^i$ are spread over four 64 bit words. For our target curves, the operand shifting requires to much efforts compared to a single multiplication on Haswell, whereas on the Sandy/Ivy Bridge this shifting pays off. Obviously, the proposed technique makes more sense the greater the hamming weight of $r(x)$ is, and if all the bits in $r(x)$ are in one 64 bit word ($s = 1$). Fortunately, this is the case for the NIST curves over $GF(2^{163})$, $GF(2^{283})$ and $GF(2^{571})$.

As a matter of fact, the performance of the `pclmulqdq` reduction strongly depends on the underlying architecture. This reduction scheme starts to pay off when the multiplier performs two multiplications faster than the according shifts and additions. Let the reduction polynomial be pentanomial with no special properties as fast shifts or other special properties. For the "shift" step of the shift&add reduction, the required amount of operations for each 128 bit word is 4×2 SHIFT's (`psllq/psrlq`) plus 3×2 XOR's. Requiring the same amount of operations for the "add" step, the multiplier must be able to perform 2 multiplications faster than this in order to beat the shift&add implementation for pentanomial reduction polynomials. On the Sandy/Ivy Bridge, this cannot be performed faster than 10 cycles (usually more), whereas **one** 64 bit multiplication has a latency of 14 cycles. This explains why this reduction scheme does not pay off on these architectures.

On Haswell however, with a latency of 7 and a throughput of only 2 cycles, this can be performed in 9 cycles. In many cases, one can make advantage of the small throughput by fetching multiple multiplications at once to gain even better performance.

5 Results

This section presents the results for the binary field arithmetic and shows the impact of the improved LD/Montgomery point multiplication and the resulting performance gain for both ECDH and ECDSA of the OpenSSL implementation. The corresponding tables can be found in the appendix. Since our implementation is suitable for several architectures, we provide data for a Haswell Core i7-4770 CPU at 3.40GHz (HSW) and a Core i5-3210M Ivy Bridge (IVB), running with 2.50GHz. Tests on a Sandy Bridge processor have produced very similar relative results as on the Ivy Bridge. To reduce randomness, we followed the guidelines from ECRYPT Benchmarking of Cryptographic Systems(eBACS) [20]. The code was compiled with the latest `gcc` in version 4.8.1. As tools for measurement we used the OpenSSL built-in speed tool as well as a cycle counter using the RDTSC instruction, similar to [10], with slightly different repetition factors. The OpenSSL speed utility can be used to reproduce our results after applying the patch and indicates the patch's impact to the overall performance of OpenSSL ECDH and ECDSA operations.

Binary field arithmetic costs are shown in Table 5. The results indicate the benefits of the fast `pclmulqdq` on Haswell and the performance difference of squaring and multiplication.

Reduction costs can be found in Table 6 on the different architectures. It shows the performance difference of trinomials and pentanomials as well as the performance loss due to the 64 bit mode in $GF(2^{239})$. Furthermore, we see the mul&add reduction outperforming the shift&add reduction with factor 1.25 in $GF(2^{163})$ and about 1.62 in $GF(2^{283})$ and 1.61 in $GF(2^{571})$ on Haswell. However, the shift&add implementation does not use AVX2 features, which could eventually allow improvements.

The computational costs for side channel protection by hiding the coordinates in order to assure a fixed memory access pattern per round are also not neclectable. A single data veiling requires twelve logical operations at the cost of an `XOR`. Additionally, one requires four temporary variables in order to hide the data processing for all four coordinates involved in the point multiplication. This has of course some performance impact to the point multiplication process, as this performance overhead applies for each loop execution.

The point multiplication's importance has been pointed out several times by now. Table 7 shows the results for $k \cdot P$ in cycles. We find that the amount of cycles has been substantially diminished, up to a factor of almost 12. Also, we observe the impact of the improved code flow for the Koblitz rather than random curves, which is noticeable in both absolute numbers and the relative speedup. Hereby, we have to mention that the numbers for the unpatched OpenSSL-1.0.1e are not constant, since it is not a constant time implementation, especially the bit length of k is not fixed. Therefore, the speedup factor is also not consistent and will differ from run to run.

Elliptic curve Diffie-Hellman operations are presented in Table 11 and Figure 2, measured with the OpenSSL speed tool and therefore the curve identifiers have been changed to `nistk` for Koblitz and `nistb` for random curves. Since the ECDH is basically a point multiplication, the speedup is very high. Prime curves are currently more frequently used in server implementations. As demonstrated in Table 8, our improvements to the point multiplication of the binary curves result in a big performance lead.

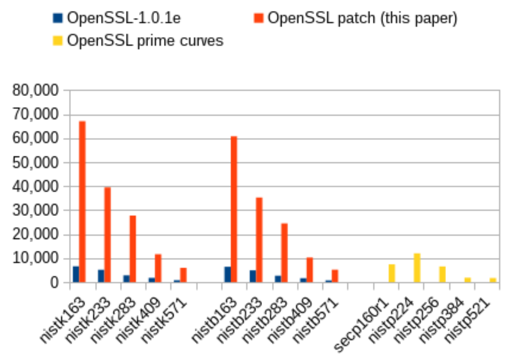

Fig. 2: ECDH - operations per second. OpenSSL (with enable-ec_nistp_64_gcc_128 flag) versus proposed implementation on the Haswell processor

Elliptic curve signature and verification algorithms have considerably more overhead than the ECDH. However, as shown in Tables 9 and 10, we perceive an average speedup of factor 4.9 (sign) and 6.2 (verify) on Ivy Bridge as well as 4.5 and 6.3 on Haswell, respectively. Since the bit length for the ECDSA in OpenSSL has been fixed *before* the point multiplication in response to a timing attack [7], the resulting scalar k is **not** in the interval $[1, n]$ anymore and is either one or two bits longer (but constant for a specific field) than the original scalar (depending on the form of the corresponding curve order n), which slightly increases the amount of cycles for the patched version.

6 Comparison to other works

Most works about the implementation of binary elliptic curves with the use of vector instructions are aiming to break speed records and thus often exploit algebraic properties. This work aims for provision of a generic point multiplication for the well known NIST/SECG curves over the binary field, without any compromises in regard to security. This work does **not** use precomputation schemes (such as wNAF, τNAF), point halving, special fields with beneficial properties (such as $\mathbb{F}_{q^2}$), or other algebraic properties (e.g., Forbenius morphism).

In Table 4 we show results for implementations of random point multiplication for binary elliptic curves in NIST/SEC fields. All numbers are given in 10^3 cycles on a single core.

	Work	SCP[1]	CPU	Method	2^{233}	2^{283}	2^{409}	2^{571}
Koblitz	Aranha et al. [2]	no	Core i7-860	4-TNAF	-	386	-	1656
	Taverne et al. [19]	no	Core i7 (SNB)	5-τNAF,τ&add	068	-	264	-
	Aranha et al. [3]	no	Core i7 (SNB)	5-τNAF,$\tau^{\lfloor 283/2 \rfloor}$	-	099	-	-
	This work	**yes**	Core i5 (IVB)	LD-Montgomery	128	217	511	1095
		yes	Core i7 (HSW)	LD-Montgomery	81	118	286	566
Random	Aranha et al. [2]	no	Core i7-860	LD-Montgomery	-	793	-	4440
	Taverne et al. [19]	no	Core i7 (SNB)	4-ωNAF	180	-	738	-
	This work	**yes**	Core i5 (IVB)	LD-Montgomery	148	254	599	1271
		yes	Core i7 (HSW)	LD-Montgomery	91	135	325	650

Table 4: Unknown Point Multiplication over NIST Curves (Koblitz)

This work yields very good results for random curves and wins against all current state-of-the-art implementations over the NIST fields, even with the costly side channel countermeasures. Since all previous works have been implemented on the Sandy/Ivy Bridge, we compare with the Ivy Bridge implementation here. For the random curve over the $GF(2^{233})$ NIST field, our implementation is about factor 1.22 (1.23 in $GF(2^{409})$) faster than [19] and even factor 3 ($GF(2^{283})$) and 3.5 ($GF(2^{571})$) faster than reported in [2] for a single core. Although our Ivy Bridge implementation for Koblitz curves beats the numbers presented in [2] by 1.77x and 1.51x, [19] achieve results which are factor 1.53 faster, whilst [3] is even as twice as fast.

[1] Side Channel Protection

For further comparison, we give the latest numbers of protected (as claimed by the authors) point multiplication implementations at the 128 bit security level, using curves with special algebraic properties. Bos et al.[6] report 117,000 cycles for a protected point multiplication on a Kummer surface with the Montgomery ladder and [16] reports 115,000 cycles on a GLS curve using 2-GLV (double&add) on a single core. At the same security level, this work achieves a point multiplication in 118,000 cycles with the OpenSSL patch [5] on the Haswell processor, without using special fields, exploiting algebraic properties or precomputation techniques. At the 256 bit security level, this work achieves a EC point multiplication in about 566,000 (Koblitz) and 650,000 (random) cycles.

7 Conclusion

In this work, we proposed a fast, constant time implementation of the point multiplication on binary elliptic curves at a security level of 80 bits and greater, standardized by NIST and SECG.

We analyzed major implementation threats for server environments and secured the implementation by applying countermeasures against common and dangerous side channel threats such as cache and remote timing attacks. Over the last years, several publications have pointed out that these side channel attacks are more than a theoretical but a serious menace. To address this issue, we provide a constant time implementation of the elliptic curve point multiplication and binary field arithmetic, without key dependent branches and a fixed memory access pattern.

Additionally, we introduced an improved version for the well known LD/-Montgomery multiplication with enhancements on the arithmetic level, provided a patch for the OpenSSL library and showed its significant performance impact. This improvement is available on all 64 bit architectures implementing at least SSE4 (better: AVX) and the `pclmulqdq` instruction. The implementation has been contributed to OpenSSL as patch for version 1.0.1e and is available for download [5].

We further showed that this impact is not limited to the point multiplication itself but improves the servers performance regarding ECDH operations and signature based cryptography. This further leads to a major performance improvement for servers and clients conducting TLS handshakes. This is an essential enhancement especially for server environments, which are often required to conduct a big number of handshakes simultaneously. Also, our results might indicate a different view to the binary ECC on server architectures.

References

1. D. Aranha, A. Faz-Hernndez, J. Lpez, and F. Rodrguez-Henrquez, *Faster implementation of scalar multiplication on koblitz curves*, in Progress in Cryptology LATINCRYPT 2012, A. Hevia and G. Neven, eds., vol. 7533 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 177–193.
2. D. Aranha, J. Lpez, and D. Hankerson, *Efficient software implementation of binary field arithmetic using vector instruction sets*, in Progress in Cryptology LATINCRYPT 2010, M. Abdalla and P. Barreto, eds., vol. 6212 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 144–161.
3. D. F. Aranha, A. Faz-Hernndez, J. Lpez, and F. Rodrguez-Henrquez, *Faster implementation of scalar multiplication on koblitz curves*. Cryptology ePrint Archive, Report 2012/519, 2012. `http://eprint.iacr.org/`.
4. D. F. Aranha, J. López, and D. Hankerson, *Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets*, in The First International Conference on Cryptology and Information Security (LATINCRYPT 2010), M. Abdalla and P. S. L. M. Barreto, eds., vol. 6212 of LNCS, 2010, pp. 144–161.
5. M. Bluhm and S. Gueron, *A fast vectorized implementation of binary elliptic curves on x86-64 processors*, 2013. `http://rt.openssl.org/Ticket/Display.html?id=3117`.
6. J. W. Bos, C. Costello, H. Hisil, and K. Lauter, *Fast cryptography in genus 2*. Cryptology ePrint Archive, Report 2012/670, 2012. `http://eprint.iacr.org/`.
7. B. B. Brumley and N. Tuveri, *Remote timing attacks are still practical*, in Proceedings of the 16th European conference on Research in computer security, ESORICS'11, Berlin, Heidelberg, 2011, Springer-Verlag, pp. 355–371.
8. P.-A. Fouque, D. Ral, F. Valette, and M. Drissi, *The carry leakage on the randomized exponent countermeasure*, in Cryptographic Hardware and Embedded Systems CHES 2008, E. Oswald and P. Rohatgi, eds., vol. 5154 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 198–213.
9. S. Gueron and M. Kounavis, *Intel carry-less multiplication instruction and its usage for computing the gcm mode*, April 2008. `http://software.intel.com/sites/default/files/article/165685/clmul-wp-rev-2.01-2012-09-21.pdf`.
10. S. Gueron and V. Krasnov, *Parallelizing message schedules to accelerate the computations of hash functions*, Journal of Cryptographic Engineering, 2 (2012), pp. 241–253.
11. T. Itoh and S. Tsujii, *A fast algorithm for computing multiplicative inverses in gf(2m) using normal bases*, Inf. Comput., 78 (1988), pp. 171–177.
12. M. Knezevic, K. Sakiyama, J. Fan, and I. Verbauwhede, *Modular reduction in gf(2^n) without pre-computational phase*, in WAIFI, 2008, pp. 77–87.
13. A. O. Krzysztof Jankowski, Pierre Laurent, *Intel polynomial multiplication instruction and its usage for elliptic curve cryptography*, 2012.
14. J. Lpez and R. Dahab, *Fast multiplication on elliptic curves over gf(2m) without precomputation*, in Cryptographic Hardware and Embedded Systems, . Ko and C. Paar, eds., vol. 1717 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1999, pp. 316–327.
15. P. L. Montgomery, *Speeding the Pollard and Elliptic Curve Methods of Factorization*, Mathematics of Computation, 48 (1987), pp. 243–264.
16. T. Oliveira, J. Lpez, D. F. Aranha, and F. Rodrguez-Henrquez, *Two is the fastest prime*. Cryptology ePrint Archive, Report 2013/131, 2013. `http://eprint.iacr.org/`.

17. Standards for Efficient Cryptography Group, *SEC 2: Recommended Elliptic Curve Domain Parameters*, in Standards for Efficient Cryptography, 2000.
18. C. Su and H. Fan, *Impact of intel's new instruction sets on software implementation of gf(2)[x] multiplication*, Inf. Process. Lett., 112 (2012), pp. 497–502.
19. J. Taverne, A. Faz-Hernndez, D. Aranha, F. Rodrguez-Henrquez, D. Hankerson, and J. Lpez, *Speeding scalar multiplication over binary elliptic curves using the new carry-less multiplication instruction*, Journal of Cryptographic Engineering, 1 (2011), pp. 187–199.
20. E. VAMPIRE, *Ecrypt benchmarking of cryptographic systems*. `http://bench.cr.yp.to/supercop.html`, 2013.
21. D. Weber and T. Denny, *The solution of mccurley's discrete log challenge*, in Advances in Cryptology CRYPTO '98, H. Krawczyk, ed., vol. 1462 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1998, pp. 458–471.

A.1 Speed results

Field	**Squaring**		**Multiplication**		**Inversion**	
	HSW	IVB	HSW	IVB	HSW	IVB
$GF(2^{163})$	21	21	34	71	4,271	4,811
$GF(2^{193})$	17	19	37	98	4,122	4,788
$GF(2^{233})$	18	21	38	87	6,074	6,956
$GF(2^{239})$	49	44	67	100	15,349	14,777
$GF(2^{283})$	27	30	53	121	11,688	9,677
$GF(2^{409})$	28	31	97	227	15,182	15,648
$GF(2^{571})$	61	68	158	335	37,118	42,595

Table 5: Cycles for arithmetic operations in $GF(2^m)$ (without addition)

Field	**Shift&Add** (Haswell)	**Mul&Add** (Haswell)	**Shift&Add** (Ivy Bridge)	**Mul&Add** (Ivy Bridge)
$GF(2^{163})$	15	**12**	18	31
$GF(2^{193})$	**11**	-	13	-
$GF(2^{233})$	**12**	-	15	-
$GF(2^{239})$	42	-	**41**	-
$GF(2^{283})$	21	**13**	25	57
$GF(2^{409})$	**15**	-	20	-
$GF(2^{571})$	37	**23**	48	81

Table 6: Comparison of the two reduction schemes on Haswell and Ivy Bridge

Binary Curve	**OpenSSL-1.0.1e** Ivy Bridge	Haswell	**OpenSSL patch** Ivy Bridge	Haswell	**Speedup factor** Ivy Bridge	Haswell
`sect163k1`	624,949	543,097	66,378	45,642	9.42	11.90
`sect163r1`	643,236	563,928	76,534	49,869	8.40	11.31
`sect163r2`	646,374	567,854	77,238	50,127	8.37	11.33
`sect193r1`	801,932	572,307	121,127	73,296	6.62	7.81
`sect193r2`	818,985	567,929	121,163	73,442	6.76	7.73
`sect233k1`	993,195	681,347	128,474	81,133	7.73	8.40
`sect233r1`	1,030,043	712,864	148,230	91,372	6.95	7.80
`sect239k1`	1,027,541	718,156	185,629	144,816	5.54	4.96
`sect283k1`	1,892,016	125,8711	217,192	118,120	8.71	10.66
`sect283r1`	2,019,134	1,341,366	254,163	134,571	7.94	9.97
`sect409k1`	3,315,895	1,984,879	510,787	286,090	6.49	6.94
`sect409r1`	3,655,279	2,142,064	599,002	325,279	6.10	6.59
`sect571k1`	7,659,288	4,764,870	1,094,765	566,257	7.00	8.41
`sect571r1`	8,432,393	5,222,559	1,270,666	659,528	6.64	7.92

Table 7: Point multiplication results in comparison to OpenSSL-1.0.1e in cycles

Curve Identifier	**Patched OpenSSL-1.0.1e** Ivy Bridge	Haswell	**Speedup factor** Ivy Bridge	Haswell
secp160r1	4444.3	7391.5	-	-
`nistk163`	27837.0	67212.4	6.26	9.09
`nistb163`	24043.5	61667.8	5.41	8.34
nistp224	7573.1	11993.8	-	-
`nistk233`	14946.4	39102.2	1.97	3.26
`nistb233`	13057.0	35246.4	1.72	2.94
nistp256	3891.5	6489.0	-	-
`nistk283`	9026.5	27586.5	2.32	4.25
`nistb283`	7754.4	24320.7	1.99	3.75
nistp384	1051.9	1848.5	-	-
`nistk409`	3879.5	11611.2	3.69	6.28
`nistb409`	3319.6	10238.1	3.16	5.54
nistp521	971.0	1682.8	-	-
`nistk571`	1822.3	5941.8	1.88	3.53
`nistb571`	1565.0	5158.8	1.61	3.07

Table 8: Comparison of ECDH operations per second for binary and prime elliptic curves on Ivy Bridge and Haswell

	Binary Curve	OpenSSL-1.0.1e		Patched OpenSSL		Speedup Factor	
		Sign	Verify	Sign	Verify		
Ivy Bridge	sect163k1	929,081	1,667,026	181,832	215,389	5.11	7.74
	sect163r1	955,187	1,743,232	190,778	238,232	5.01	7.32
	sect163r2	959,119	1,734,078	193,299	236,150	4.96	7.34
	sect193r1	1,004,362	1,806,773	248,191	335,012	4.05	5.39
	sect193r2	1,007,317	1,816,476	248,683	334,215	4.05	5.44
	sect233k1	1,161,852	2,125,398	262,615	362,315	4.42	5.87
	sect233r1	1,197,683	2,226,927	289,039	405,330	4.14	5.49
	sect239k1	1,194,324	2,210,181	321,347	482,566	3.72	4.58
	sect283k1	2,097,257	4,006,091	366,470	557,568	5.72	7.18
	sect283r1	2,227,001	4,241,617	404,091	628,606	5.51	6.75
	sect409k1	3,505,242	6,801,755	702,767	1,181,176	4.99	5.76
	sect409r1	3,776,632	7,322,093	793,073	1,359,816	4.76	5.38
	sect571k1	8,018,930	15,635,228	1,365,881	2,461,485	5.87	6.35
	sect571r1	8,745,813	17,133,137	1,542,300	2,816,402	5.67	6.08
Haswell	sect163k1	680,085	1,215,819	157,761	173,166	4.31	7.02
	sect163r1	698,430	1,256,375	160,901	180,300	4.34	6.97
	sect163r2	702,585	1,246,757	163,610	182,937	4.29	6.82
	sect193r1	730,894	1,302,676	198,160	234,577	3.69	5.55
	sect193r2	732,172	1,306,010	197,911	236,197	3.70	5.53
	sect233k1	841,066	1,530,004	213,628	264,756	3.94	5.78
	sect233r1	875,737	1,585,333	223,396	285,478	3.92	5.55
	sect239k1	869,989	1,582,703	278,374	402,386	3.13	3.93
	sect283k1	1,440,771	2,724,199	266,409	355,636	5.41	7.66
	sect283r1	1,517,952	2,882,340	283,443	386,729	5.36	7.45
	sect409k1	2,201,747	4,230,856	480,983	738,817	4.58	5.73
	sect409r1	2,353,768	4,524,217	524,003	822,021	4.49	5.50
	sect571k1	5,091,193	9,885,973	833,567	1,384,070	6.11	7.14
	sect571r1	5,478,258	10,727,383	914,464	1,552,640	5.99	6.91

Table 9: ECDSA sign and verify cycles for the NIST curves on Ivy Bridge and Haswell

	Binary Curve	OpenSSL-1.0.1e		OpenSSL patch		Speedup factor	
		Ivy Bridge	Haswell	Ivy Bridge	Haswell	Ivy Bridge	Haswell
SIGN	nistk163	3,749.9	6,465.3	17,721.8	36,872.6	4.73	5.70
	nistk233	1,881.7	3,259.2	10,359.0	22,998.4	5.51	7.06
	nistk283	1,267.5	2,204.7	6,688.9	16,884.9	5.28	7.66
	nistk409	542.2	977.0	3,140.9	8,150.0	5.79	8.34
	nistk571	257.6	466.4	1,556.0	4,424.1	6.04	9.49
	nistb163	3,766.5	6,487.3	16,203.5	35,110.0	4.30	5.41
	nistb233	1,893.1	3,279.2	9,386.5	21,468.8	4.96	6.55
	nistb283	1,265.7	2,196.4	5,962.3	15,602.7	4.71	7.10
	nistb409	539.3	976.3	2,763.4	7,423.1	5.12	7.60
	nistb571	257.2	466.6	1,354.8	3,977.0	5.27	8.52
VERIFY	nistk163	1,578.6	3,159.5	11,688.1	26,508.4	7.40	8.39
	nistk233	1,211.6	2,419.8	6,439.4	15,557.1	5.31	6.43
	nistk283	639.3	1,355.7	3,951.1	11,003.2	6.18	8.12
	nistk409	361.9	839.1	1,757.1	4,845.0	4.86	5.77
	nistk571	159.9	368.3	834.6	2,533.6	5.22	6.88
	nistb163	1,514.5	3,043.9	10,453.8	24,904.8	6.90	8.18
	nistb233	1,150.4	2,348.0	5,711.9	14,095.6	4.97	6.00
	nistb283	594.2	1,283.5	3,445.5	9,888.5	5.80	7.70
	nistb409	344.2	786.9	1,522.4	4,361.9	4.42	5.54
	nistb571	145.7	341.0	724.9	2,251.6	4.98	6.60

Table 10: OpenSSL ECDSA sign and verification operations per second for the NIST curves on Ivy Bridge and Haswell

Binary Curve	OpenSSL-1.0.1e		OpenSSL patch		Speedup factor	
	Ivy Bridge	Haswell	Ivy Bridge	Haswell	Ivy Bridge	Haswell
nistk163	4,129.0	6,586.9	34,739.0	67,029.6	8.41	10.18
nistk233	2,628.4	5,121.9	17,013.4	39,441.3	6.47	7.70
nistk283	1,338.5	2,825.7	9,017.6	27,718.5	6.74	9.81
nistk409	836.3	1,745.8	3,884.0	11,634.2	4.64	6.66
nistk571	406.5	763.2	1,824.1	5,930.9	4.49	7.77
nistb163	3,217.6	6,382.5	24,237.2	60,729.6	7.53	9.52
nistb233	2,443.5	4,881.9	13,114.3	35,230.4	5.37	7.22
nistb283	1,244.3	2,651.6	7,711.6	24,456.4	6.20	9.22
nistb409	723.4	1,640.3	3,382.3	10,228.6	4.68	6.24
nistb571	368.3	693.8	1,943.5	5,172.1	5.28	7.45

Table 11: OpenSSL ECDH operations per second over the NIST curves

A.2 New 2P-Algorithm subfunctions

Algorithm 2: Point Doubling algorithm for random curves

```
Input: $c \in GF(2^m)$ where $c^2 = b$, $x$-coordinate $X/Z$ for a point $P$
Output: $x$-coordinate $X/Z$ for the point $2P$
1  $t \leftarrow c$;
2  $X \leftarrow X^2 \bmod f(x)$;
3  $Z \leftarrow Z^2 \bmod f(x)$;
4  $t \leftarrow Z \times t \bmod f(x)$;
5  $Z \leftarrow Z \times X \bmod f(x)$;
6  $X \leftarrow X + t$;
7  $X \leftarrow X^2 \bmod f(x)$;
8  return $X$, $Z$
```

Algorithm 3: Point Doubling algorithm for Koblitz curves

```
Input: $x$-coordinate $X/Z$ for a point $P$
Output: $x$-coordinate $X/Z$ for the point $2P$
1  $X \leftarrow X^2 \bmod f(x)$;
2  $Z \leftarrow Z^2 \bmod f(x)$;
3  $t \leftarrow X + Z$;
4  $Z \leftarrow Z \times X \bmod f(x)$;
5  $X \leftarrow t^2 \bmod f(x)$;
6  return $X$, $Z$
```

Algorithm 4: Algorithm for Point Addition

```
Input: $x$-coordinates $X/Z$ of the point $P(x,y), P_0(X_0,Z_0), P_1(X_1,Z_1)$
Output: $x$-coordinate $X_1/Z_1$ for the point $P_0 + P_1$
1  $t_0 \leftarrow X_1$;
2  $X_0 \leftarrow X_0 \times Z_0 \bmod f(x)$;
3  $Z_0 \leftarrow Z_0 \times t_0 \bmod f(x)$;
4  $t_1 \leftarrow X_0 \times Z_0$ ;          // mult. w/o reduction, result double sized
5  $Z_0 \leftarrow Z_0 + X_0$;
6  $Z_0 \leftarrow Z_0^2 \bmod f(x)$;
7  $t_0 \leftarrow x$;
8  $t_2 \leftarrow Z_0 \times t_0$ ;          // mult. w/o reduction, result double sized
9  $t_2 \leftarrow t_2 + t_1$ ;                                 // double sized addition
10 $X_0 \leftarrow t_2 \bmod f(x)$ ;                                  // final reduction
11 return $X_0$,$Z_0$
```