J Cryptogr Eng (2015) 5:141–151
DOI 10.1007/s13389-014-0090-x

REGULAR PAPER

Fast prime field elliptic-curve cryptography with 256-bit primes

Shay Gueron · Vlad Krasnov

Received: 3 December 2013 / Accepted: 25 October 2014 / Published online: 17 November 2014

Abstract This paper studies software optimization of elliptic-curve cryptography with 256-bit prime fields. We propose a constant-time implementation of the NIST and SECG standardized curve P-256, that can be seamlessly integrated into OpenSSL. This accelerates Perfect Forward Secrecy TLS handshakes that use ECDSA and/or ECDHE, and can help in improving the efficiency of TLS servers. We report significant performance improvements for ECDSA and ECDH, on several architectures. For example, on the latest Intel Haswell microarchitecture, our ECDSA sign is 2.33× faster than OpenSSL's implementation.

Keywords ECDSA · ECDH · EC · SSL · TLS · Optimization · Haswell

1 Introduction

TLS [7] is the leading protocol for secure network communications. It supports a variety of symmetric ciphers and MAC algorithms for the authenticated and encrypted client-server communication, and a variety of public key algorithms for establishing a symmetric session key (for the authenticated encryption).

Currently, the most popular key-exchange algorithm is based on RSA. Here, the client generates a secret value (master key) and encrypts it using the server's public RSA key. The server decrypts that value, and both parties apply some agreed Key Derivation Function to derive the session key.

With this approach, the confidentiality of all the sessions depends on the server's private RSA key: the confidentiality of any session, with any client (even past recorded sessions) is lost if this key is compromised (e.g., lost, hacked, subpoenaed). In response to increased sensitivity to this property (e.g., due to information on the PRISM projects [11][1]), some bodies and companies are migrating to a "Perfect Forward Secrecy" (PFS) protocol (e.g., [19]), and some have already implemented it (e.g., [14]). Here, the server-client key exchange uses ephemeral parameters, rather than a single fixed key. Two such key exchange algorithms are supported by TLS: Ephemeral Diffie–Hellman (DHE), and Elliptic-Curve Ephemeral Diffie–Hellman (ECDHE). These algorithms require the server and the client to (randomly) select a secret key, and to use it for generating and exchanging "public key parameters". These are subsequently used for deriving a shared session key. In such protocols, the server is required to authenticate itself (to the client) by signing the parameters that it sends to the client. Several signature algorithms can be agreed during the initial client-server handshake, and the two leading ones are RSA signature and elliptic-curve digital signature algorithm (ECDSA). Consequently, adding support for PFS protocols makes DHE+RSA, ECDHE+RSA and ECDHE+ECDSA key exchange and signature combinations important targets for optimization, especially for servers.

S. Gueron (✉)
Department of Mathematics, University of Haifa, Haifa, Israel
e-mail: shay@math.haifa.ac.il

S. Gueron · V. Krasnov
Intel Corporation, Israel Development Center, Haifa, Israel

[1] "Demand for encryption apps has increased dramatically ever since the exposure of massive internet surveillance programs run by US and UK intelligence agencies. Now Facebook is reportedly moving to implement a strong, decades-old encryption technique that's been largely avoided by the online services that need it most"; J. Kopstein, The Verge, http://www.theverge.com/2013/6/26/4468050/facebook-follows-google-with-tough-encryption-standard.

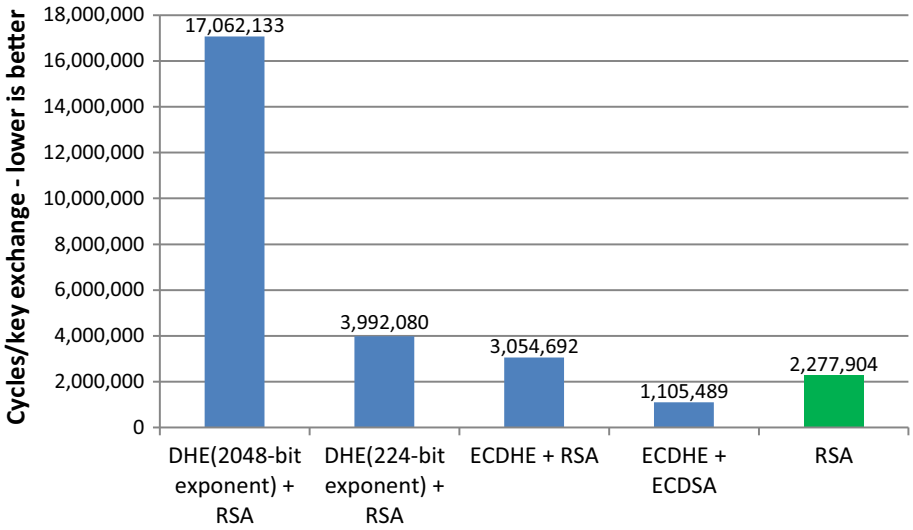

Fig. 1 The performance of PFS supporting algorithms, measured in CPU cycles on the latest Intel Haswell microarchitecture. The RSA2048 and DH2048 numbers are based on the currently fastest implementation [8]. DH2048 is shown with both 2,048-bit exponent (OpenSSL's default) and with 224-bit exponent (that suffices for 112 bits of security). For ECDSA and ECDHE, the numbers correspond to the NIST P-256 curve (OpenSSL development version of OpenSSL; September 9, 2013, configured with 'enable-ec_nistp_64_gcc_128'). Note that the ECDHE+ECDSA combination provides 128 bits of security, and the other combinations provide only 112 bits. For comparison, the rightmost (*green*) *bar* shows the performance of the RSA-based key exchange, that does not provide PFS

From the server's viewpoint, migration to ECC-based TLS connections can be done in two ways. Adopting the full ECDHE+ECDSA combination requires the server to use an EC certificate. On the other hand, the ECDHE+RSA combination allows a server to continue using an existing RSA certificate, and use ECC only for the key exchange. Direct browsing indicates that current ECC adopters indeed use both combinations. Few examples are Google (using ECDHE+RSA and ECDHE+ECDSA), Facebook and Twitter (recently adopting ECDHE+RSA).

The currently recommended RSA key size is 2048-bit, and it is estimated to provide 112 bits of security [2] (for 128 bits security, one needs RSA3072, which computationally is ~ 3.375 times more expensive (measurements of the development version of OpenSSL, show that RSA3072 sign is 4.64 times slower). The computational cost of RSA2048 can be estimated as the cost of two 1024-bit modular exponentiations. The DHE protocol (providing 112 bits of security) requires two modular exponentiations with a 2048-bit modulus, and it suffices to use a 224/256 bit exponent. With these parameters, the performance of DHE is comparable to that of RSA. Elliptic-curve algorithms for signature and key exchange require shorter keys of 224/256 bits for 112/128 bits of security. With such keys, ECDSA signatures[2] and ECDH are significantly faster than RSA signatures and DH key exchange counterparts. Figure 1 shows the performance of these signature and key exchange algorithms, with a comparison to the classical non-PFS key exchange based on RSA alone. It shows that supporting PFS key exchange, with DHE+RSA and ECDHE+RSA combinations, comes with a performance cost, but the ECDHE+ECDSA combination is actually faster than the non-PFS key exchange.

Elliptic-curve cryptography with a 224-bit prime (NIST P-224 curve) has been recently optimized by [13], contributed to OpenSSL, and is now part of its current offering. Subsequently, a similar optimized implementation was derived from [13], to support the 256-bit and 521-bit NIST primes.[3] These optimizations provide significant speedups compared to original OpenSSL implementations. However, as NSA's Suite B endorses only 256 and 384-bit prime curves ECC [20], the de-facto standard that is adopted on the web, uses the 256-bit prime.

While ECC can be used with any prime, NIST specifies one prime-field curve for each of the sizes of 192, 224, 256, 384 and 521 bits [16]. These are "generalized Mersenne" primes [21], where modular reduction can be implemented efficiently.

This paper studies software optimizations for ECC with 256-bit primes (NIST P-256 curve in particular). We apply

[2] RSA signature verification with the standard short public exponent remains faster than ECDSA verification. However, verification is done by the client, and not by the server side.

[3] An optimized implementation of P-224, P-′256 and P-521 was contributed to OpenSSL by Emilia K sper, Adam Langley and Bodo Moeller. To enable it, OpenSSL should be configured with 'enable-ec_nistp_64_gcc_128'.

Point Doubling

Input: (X, Y, Z)

$S = 4XY^2$; $M = 3X^2 + aZ^4$

$X' = M^2 - 2S$

$Y' = M(S - X') - 8Y^4$

$Z' = 2YZ$

Output: (X', Y', Z')

Point Addition

Input: (X_1, Y_1, Z_1), (X_2, Y_2, Z_2)

$U_1 = X_1Z_2^2$; $U_2 = X_2Z_1^2$; $S_1 = Y_1Z_2^3$; $S_2 = Y_2Z_1^3$

```
if (U1 == U2) then
  if (S1 != S2)  return POINT_AT_INFINITY
  else return POINT_DOUBLE (X1, Y1, Z1)
  abort
end
```

$H = U_2 - U_1$; $R = S_2 - S_1$

$X_3 = R^2 - H^3 - 2U_1H^2$

$Y_3 = R(U_1H^2 - X_3) - S_1H^3$

$Z_3 = HZ_1Z_2$

Output: (X3, Y3, Z3)

Fig. 2 Point doubling and point addition in Jacobian coordinates. The arithmetic operations are in the underlying field *GF(p)*. A constant time implementation would perform the operations and resolve the conditional statements in a constant time manner

our proposed optimizations to ×86–64 architectures, but any 64 or 32 bit architecture may potentially benefit from them as well. Our implementation includes side channel protection up to the protocol level, and we use several optimizations for improving the performance. In particular, we propose a method for implementing ECC in the Montgomery domain, and optimize it to what we call "Montgomery Friendly" primes. The NIST P-256 curve has such a prime.

2 Preliminaries

Prime field elliptic curves are defined by the pairs (x, y) satisfying the relation $y^2 = x^3 + ax + b$ with $a, b \in$ GF(p) (satisfying $4a^3 + 27b^2 \neq 0$) and where p>3 is a prime. For any two points P, Q on a given curve, their addition $(P + Q)$ is defined as a fundamental operation (using the tangent-and-chord rule), and this defines a group of points on the curve. The addition $P + Q$ when $P \neq$ Q is called point addition, and addition $P + Q = 2P$ when $P = Q$ is called point doubling (and they are different operations). Consequently, for non-negative integer k, it is possible to define the scalar point multiplication $k \cdot P$ on the curve.

In this paper, we consider the two elliptic-curve cryptosystems (ECC) mentioned above, namely ECDHE and ECDSA. Their security relies on the difficulty of the elliptic-curve discrete logarithm problem, i.e., finding the value of k, when $k \cdot P$ is given. Obviously, implementation of these protocols involves (among other computations) the computation of point multiplications over the given field.

For the sake of optimization, the EC affine coordinates (x, y) can be converted to a representation where the group operations are cheaper (specifically, involve fewer *GF(p)* inversions). We use here the Jacobian point representation with three coordinates (X, Y, Z), where $x = X/Z^2$, $y = Y/Z^3$ (modulo the relevant prime). The point doubling and point addition operations in these coordinates are provided in Fig. 2.

By definition, converting the triplet (X, Y, Z) from Jacobian back to affine coordinates requires field inversion(s). This conversion needs to be carried out only once, at the end of the computation of the point multiplication $k \cdot P$.

Generalized Mersenne [21] primes are of special interest. They are defined as primes of a special form that allows for efficient reduction, without integer-division. Generalized Mersenne primes are frequently used for ECC (all the NIST primes in [16] are such) because reduction modulo of such primes can be carried out efficiently. p_{256} is a generalized Mersenne prime (see Fig. 4).

Figure 3 illustrates the ECDH and ECDSA flows (note that during the TLS handshake, the server computes an ECDSA signature).

In the rest of the paper, we consider only the case $a = -3$ mod p, and primes whose bit-length is 256 (i.e., satisfying $2^{255} < p < 2^{256}$).

2.1 The 256-bit NIST curve's parameters

The results we show in this paper relate to optimizations for the NIST 256-bit curve, P-256, with a prime denoted by p_{256}. The related parameters are given in Fig. 4.

Fig. 3 The flows for ECDSA signature and ECDH key exchange

The (public) curve parameters are:
a, b, p (prime), G (the generator point), n (the multiplicative order of G).
The private data: d_s – server secret
The signed data: z – hash(message) truncated to len(n)

ECDSA signature

Pick a random integer k∈[1,n-1]
Compute $(x_1, y_1) = k \cdot G$
$r = x_1 \bmod n$
$s = k^{-1}(z + rd_S) \bmod n$
The signature is the pair (r, s)

ECDH

Server picks a random integer $d_S \in [1,n-1]$
Client picks a random integer $d_C \in [1,n-1]$
Generate key:
    Server generates it's public key $Q_S = d_S \cdot G$
    Client generates it's public key $Q_C = d_C \cdot G$
Compute key:
    Server computes $(x_1, y_1) = d_S \cdot Q_C$
    Client computes $(x_1, y_1) = d_C \cdot Q_S$
The shared secret is x_1

Fig. 4 The NIST 256-bit curve parameters [15]

$p_{256} = 2^{256}-2^{224}+2^{192}+2^{96}-1$
a = p-3
The base point G:
x_G =
0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296
y_G =
0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5
the order of G, *n* =
0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551

3 A Montgomery-friendly modulus

The Montgomery multiplication [15] (*MM* hereafter) is a well-known efficient technique for computing modular exponentiation [9]. Similarly, it can be used for other modular arithmetic intensive algorithms, such as the elliptic-curve point multiplication.

Point doubling and addition (see Fig. 2) involves a sequence of operations modulo the prime p. It is possible to execute them in the so-called Montgomery domain, where the modular multiplication $A \times B \bmod p$ is replaced by the Montgomery Multiplication operation, $MM(A, B) = A \times B \times 2^{-l} \bmod p$, and where the positive integer l is some parameter. When $A = B$, we call the operation a Montgomery Square (*MSQR*), and this case can be optimized by leveraging the fact that $A = B$.

To compute a point multiplication using *MM* operations, the coordinates of the input point *(X, Y, Z)* need to be converted to the Montgomery domain. This is done by multiplying (modulo p) each coordinate by 2^l, or alternatively, performing an *MM* by the constant $H = 2^{2l} \bmod p$. After all the computations are completed (in the Montgomery domain), they need to be converted back to the residue domain. This is done by *MM* by 1. The Word-by-Word flow for computing *MM* (*WW-MM*) is described in Fig. 5 (left panel).

For some cases, such as the one we discuss here, a shortcut is available, and to this end, we use the following definition.

Definition 1 let p be an odd modulus and s be a positive integer. If p satisfies $-1/p \bmod 2^s = 1$, then p is called an s-Montgomery Friendly modulus (MF for short).

For an MF modulus, the *WW-MM* computations can be optimized, as shown in Fig. 5 (right panel).

Our study discusses a modulus which is a 256-bit prime, implying $l = 256$. To optimize code for 64-bit architectures, we use $s = 64$. This implies that $k = 4$. In other words, each

Fig. 5 *Left panel*: word by word Montgomery multiplication (*WW–MM*). *Right panel WW–MM* for a Montgomery friendly modulus. In our context, the relevant parameters for a 256-bit prime modulus (p) are $l = 256$, $s = 64$, and $k = 4$

```
Algorithm 1: Word-by-Word
Montgomery Multiplication (WW-MM)
Input: p < 2^l (odd modulus),
       0 ≤ a, b < p, l=s×k
Output: a×b×2^-l mod p
Pre-computed: k0 = -p^-1 mod 2^s
Flow
  1. T = a×b
   For i = 1 to k do
     2. T1 = T mod 2^s
     3. Y = T1 × k0 mod 2^s
     4. T2 = Y × p
     5. T3 = (T + T2)
     6. T = T3 / 2^s
   End For
  7. If T ≥ p then X = T - p;
        else X = T
Return X
```

```
Algorithm 2: Word-by-Word
Montgomery Multiplication for a
Montgomery Friendly modulus p
(Montgomery Friendly
Multiplication)
Satisfying -p^-1 mod 2^s=1.
Input: p < 2^l
     (Montgomery Friendly modulus)
     0 ≤ a, b < p, l=s×k
Output: a×b×2^-l mod p
Flow
  1. T = a×b
   For i = 1 to k do
     2. T1 = T mod 2^s
     3. T2 = T1 × p
     4. T3 = (T + T1)
     5. T = T3 / 2^s
   End For
  6. If T ≥ p then X = T - p;
        else X = T
Return X
```

field element is considered as a "4 digits number", where each digit has 64 bits.

In the general case, where an l-bit number can be represented using k digits, *MM* requires $2k^2 + k$ single-precision multiplication operations. Optimizing for an MF prime reduces the number of single-precision multiplications to $2k^2$. With $k = 4$, as in our case, this saves more than 10 % of the operations (for a large k, the saving can be insignificant).

A similar optimization is employed in the NSS [17] project, for $s = 32$.

Observation 1 A 256-bit MF modulus q has the form $q = 2^{255-offset} \times L + 2^{64} - 1$, or the form $q = 2^{255-offset} \times L - 1$, where offset = floor ($\log_2$ (L)) (and it is bounded appropriately, to assure that $q < 2^{256}$).

Example 1 The first *256*-bit MF prime of the first form is $2^{250} \times 61 + 2^{64} - 1$. The first *256*-bit MF prime of the second form is $2^{247} \times 321 - 1$.

Observation 2 The NIST prime p_{256} is a MF prime.

Observation 3 Another property of p_{256}, can be used for optimization in Step 3 (on the right pane of Fig. 5). We use it to optimize the multi-precision multiplication by p_{256}. When writing p_{256} in hexadecimal notation (64-bit "quad-words"), we note that quad-word 0 is: 0xffffffffffffffff. Since $T1 \times 0xffffffffffffffff = T1 \times 2^{64} - T1$, it follows that the multiplication by this quad-words can be replaced by a (quicker) subtraction. In addition, quad-word 2 of p_{256}, is 0 (i.e., no multiplication is needed at all). These optimizations lead to a performance gain of 1.07× for sign and 1.14× for verify.

4 Point doubling and point addition in the Montgomery domain

The operations in Montgomery domain correspond to operations in the Residue domain as follows: if *OP(a, b)* is the operation in the Residue domain, it corresponds to $MOP(a \times 2^{-n}, b \times 2^{-n}) = OP(a, b) \times 2^{-n}$ in the Montgomery domain. Unlike modular-exponentiation that has only multiplications, Point Addition and Point Doubling involve addition, subtraction, and also multiplications by some small constants. These can be performed using regular modular (modulo p) operations:

$$\text{Addition} : a \times 2^{-n} + b \times 2^{-n} = (a + b) \times 2^{-n} \pmod{p}$$
$$\text{Subtraction} : a \times 2^{-n} - b \times 2^{-n} = (a - b) \times 2^{-n} \pmod{p}$$
$$\text{Multiplication by constant} c : c \times (a \times 2^{-n}) = (c \times a) \times 2^{-n} \pmod{p}$$

The relevant constants for point-doubling are 2, 4, 8, and 3. However we use an equivalent flow that requires multiplication by 2 and 3, and a division by 2. Multiplication and division by two is implemented using bit shift to the left/right, and a constant time conditional subtraction/addition of the modulus, similarly do the addition and subtraction operations. Multiplication by 3 is implemented as multiplication by two followed by an addition.

5 Optimizing at the protocol level

This section describes the methods we applied in order to optimize the ECDSA and the ECDH at the protocol level.

5.1 The four computational problems for optimizing ECDSA and ECDH

ECDSA and ECDH implementations include four independent algorithmic flows: ECDSA sign, ECDSA verify, ECDH generate key, ECDH compute key. The efficiency of these flows depends on optimization of the following four computational problems (CP hereafter):

CP 1: multiply the point G (the generator) by a scalar.
CP 2: multiply an arbitrary point P by a scalar.
CP 3: extract the affine coordinate x (and y). We use here the Jacobian coordinates, so the back-conversion involves a 256 bit modular inverse (modulo p_{256}).
CP 4: Compute a 256-bit modular inverse, with the modulus n = Order(G).

These problems are used in the following combinations:

- ECDSA sign uses: CP1, CP3 and CP4.
- ECDSA verify uses: CP1, CP2, CP3 and CP4.
- ECDH generate key uses: CP1 and CP3.
- ECDH compute key uses: CP2 and CP3.

6 The different components of the optimization

Obviously, CP1 and CP2 dominate the computations, so they are the first optimization target. For CP1, we note that the generator (G) is part of the pre-defined curve parameters (i.e., fixed). We implement the scalar multiplication (by G) with a windowing method and Booth encoding [4], using a window of size 7, and avoid *MSQR*'s via pre-computation. There are $[256/7] = 37$ windows, and hence 37×2 tables (storing the X and the Y coordinate; Z is implicitly 1). The values stored in the tables are

$$\text{Table}[i][j] = 2^{7i} \times (j \times G) (\text{mod } p_{256})$$

Each table is stored, and fetched in a side-channel protected manner: accessing a table does not employ memory access patterns that depend on the (secret) scalar, instead, for a given window all elements are loaded sequentially, and masked based on the secret window value. The mask is such, that all values but the required one are zeroed. To optimize this flow we make use of SIMD instructions.

By using such large tables, the computation of CP1 requires only 36 point additions, and no point-doublings. The additions can be further optimized and performed in parallel using SIMD instructions. For the Intel® CPU codename Haswell, we wrote such a code that performs 4 point additions in parallel using AVX2 [12] instructions. In the end the 4 independent results are summed serially.

The pre-computed (fixed) tables require slightly more than 150KB of storage, which exceeds the size of the first level cache in modern CPU's. However, our tests indicate that using these large tables yields better performance than the performance obtained with smaller tables. For completeness, our implementation includes a function that generates the tables, but a server can choose to just hold a static-fixed value for P-256.

Note that for this usage, a server needs a single table for all the connections (unlike RSA's modular exponentiation computations, where the tables depend on the exponentiation base, and are different for each connection).

To speed up point multiplication (for a general point P and also by G) we wrote *MM* and *MSQR* assembly routines that are specifically optimized for the MF prime p_{256}. We also use windowing method with Booth encoding, but with a smaller window size of 5.

The aggressive optimization of CP1 and CP2 increases the relative weight of CP3, which becomes the next optimization target. To implement modular inversion (modulo p_{256}) we used the Fermat's Little Theorem and wrote a dedicated modular-exponentiation function (optimized for the generalized Merssene p_{256}). It computes the modular inverse at the cost of 255 *MSQR*'s and 13 *MM*'s. When only the x coordinate is required, thus only $Z^{-2} (\text{mod}\, p)$ needs to be computed, we can reduce the number of *MM*'s to 12.

We did not prepare a dedicated function for CP4, because n is neither a Pseudo-Mersenne nor a MF prime. Instead, we used the already highly-optimized constant time implementation of Montgomery modular exponentiation, that is present in the OpenSSL (this implementation was recently improved by incorporating some contributed ideas from [9]).

Some implementations of the verify routine perform a double-scalar multiplication, fusing the computation of CP1 and CP2, in our case however, CP1 is optimized to the point where the double-scalar multiplication does not gain performance.

7 Side channel protection

Cryptographic software implementations are nowadays expected to be protected against side channel leakage. Specifically, a protected implementation must not include branches, memory access patterns and instructions flows that depend on secret information.

Side-channel information in an ECC implementation can potentially leak from two sources. The first is the scalar multiplication $k \cdot P$, where k is secret (i.e., a key). This happens if the implementation accesses tables in a way that depends on the value of k, or if it has branches that depend on k. However, we point out that closing this potential leak is not sufficient for generating a fully protected ECDSA implementation: the

step $(z + rd_S)k^{-1} mod\, n$ (see Fig. 3) can also leak secret information if the modular inverse $1/k \bmod n$ is computed in an unsafe manner.

The recent optimized implementation [13] and the subsequently derived implementation for P-256, have a constant-time point multiplication. By comparison, our implementation addresses both side channel leak sources via constant-time point multiplications, as well as constant time modular inversion. To this end, we modified the OpenSSL *ecdsa_sign_setup* routine so that it uses OpenSSL's constant time modular exponentiation function based on Fermat's Little Theorem (rather than the original OpenSSL *BN_mod_inverse* that is based on Extended Euclidean Algorithm and is implemented with branches that can potentially give away k; see [1] for details discussing this leak). As a result, our software patch makes the entire ECDSA sign function constant time.

We note that the verification routine does not require constant-time operation, and this overhead can be removed from this routine. However, we did not think that the potential performance gain justifies adding a non-constant-time function to the verification code.

8 Results

This section details the performance results that we obtain through our proposed optimizations. For that purpose we wrote a patch for OpenSSL using the x86-64 assembly language [10]. We compare it to the performance of the latest development branch of OpenSSL (from September 9, 2013; retrieved from the "git" repository of OpenSSL [18]). We configured OpenSSL with both the default parameters (denoted 'OpenSSL default' here), and with 'enable-ec_nistp_64_gcc_128' (denoted 'OpenSSL NISTP enabled').

Figure 6 shows the performance gains in CP1, CP2, CP3 (see Sect. 5 for the definitions). The computation of $k.\ G$ gains a speedup factor of more than 3× on the new Haswell microarchitecture. Most of this gain can be attributed to the pre-computations that we use. Recall that the computation of $k \cdot G$ is used for both ECDSA signature and ECDHE generate key routines. These are computed by the server side during a TLS handshake. We point out that the OpenSSL default configuration, is not side-channel protected.

The $k \cdot P$ computation shows significant speedup as well: on the Haswell microarchitecture it is 1.8× times faster than the optimized implementation of OpenSSL NISTP. This performance gain is due to our *MM* and *MSQR* implementations that are also optimized for the MF p_{256} prime.

Finally, conversion from Jacobian to affine coordinates is sped up by a factor of 2.11× on the Haswell microarchitecture, thanks to using optimized *MM* and *MSQR* routines and a reduced number of operations (using $(p_{256}) - 2$ as the exponent).

Figures 7 and 8 demonstrate the performance, at the protocol level, of ECDSA signing, verifying, and the ECDH Compute Key. This performance is measured via OpenSSL's built-in benchmarking utility "OpenSSL speed". Note that this utility benchmarks only the Compute Key part of the ECDH key exchange (and not the Generate Key part), although in a real application, the ECDH parameters are ephemeral, and the Generate Key routine is also performed for every connection.

For ECDSA signature, we point out that only our implementation is entirely constant-time. For ECDH, both ours and OpenSSL's optimized NISTP implementations are constant-time.

On a Haswell CPU, in a single threaded run, our implementation is 2.33× faster for ECDSA sign, 1.86× faster for ECDSA verify, and 1.8× faster for the key computation. The

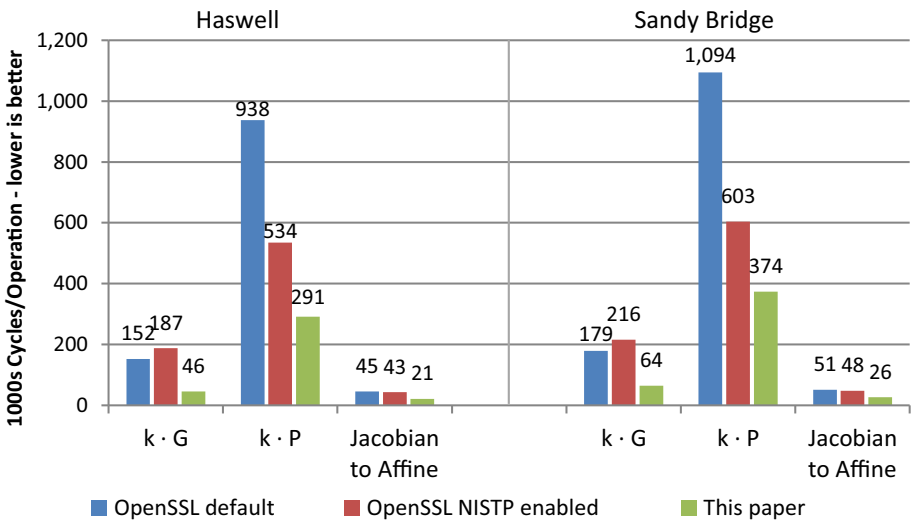

Fig. 6 The performance (in 1,000 s of CPU cycles) on the NIST P-256 curve, of the fundamental EC operations $k \cdot G$, $k \cdot P$ and Jacobian-to-affine coordinates transformation. The *left panel* shows the performance on the Haswell microarchitecture, and the *right panel* shows the performance on the Sandy Bridge microarchitecture

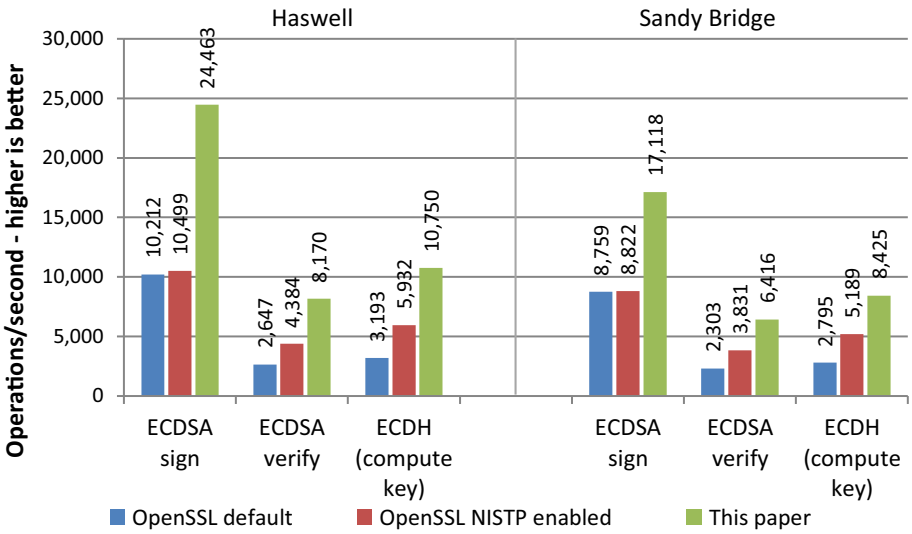

Fig. 7 The performance of ECDSA sign, ECDSA verify, and ECDE Compute key, measured by 'OpenSSL speed', for P-256, at the frequency of 3.4 GHz, for a single core, and a single threaded run. The *left panel* shows the performance on the Haswell microarchitecture, and the *right panel* shows the performance on the Sandy Bridge microarchitecture

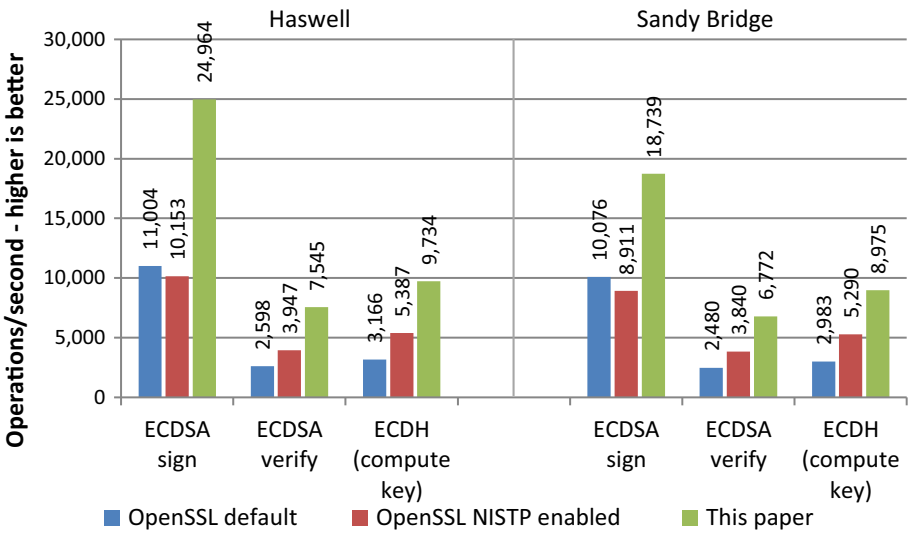

Fig. 8 The performance of ECDSA sign, ECDSA verify, and ECDE Compute key, measured by 'OpenSSL speed', for P-256, at the frequency of 3.4 GHz, for a single core, and a Hyper Threaded run. The *left panel* shows the performance on the Haswell microarchitecture, and the *right panel* shows the performance on the Sandy Bridge microarchitecture

hyper-threaded performance shows an even larger gap, with the respective speedup factors of 2.46×, 1.91× and 1.81× (all compared to OpenSSL's optimized NISTP implementation).

8.1 System level effects

We show here the impact of our optimizations at the system level. We measured the actual performance of a Haswell-based server machine, running on single core at 1GHz, using the ECDHE-ECDSA-AES128-GCM-SHA256 cipher suite (this is the cipher suite selected when connecting to Gmail service with the Chrome browser version 31). We defined the session to download a 500KB file. More details on the configuration used for the experiment are described in Appendix A.

By applying our patch, the server improved its throughput by 1.22× (from ∼300 to ∼367 connections/second), compared to the original OpenSSL-based connection. Figure 9

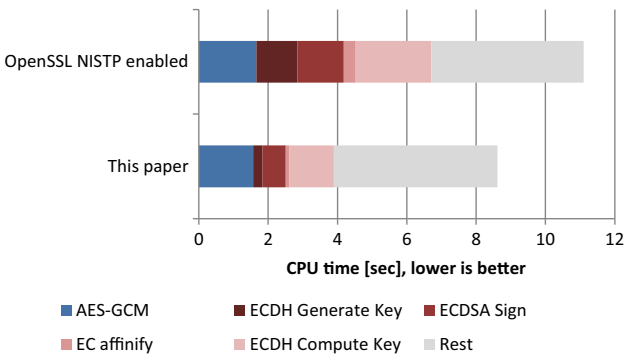

Fig. 9 The time (and its breakdown) to service 2,000 TLS connections, each one downloading a 500 KB file. The experiment uses an Apache server profiling tool ("ab"), and is run on a single Haswell core that is down-clocked to 1 GHz (hyper-threading enabled). The *graph* compares the development version of OpenSSL to a system with OpenSSL patched with our improved software. It shows that our improvements make the server's throughput 1.2× higher

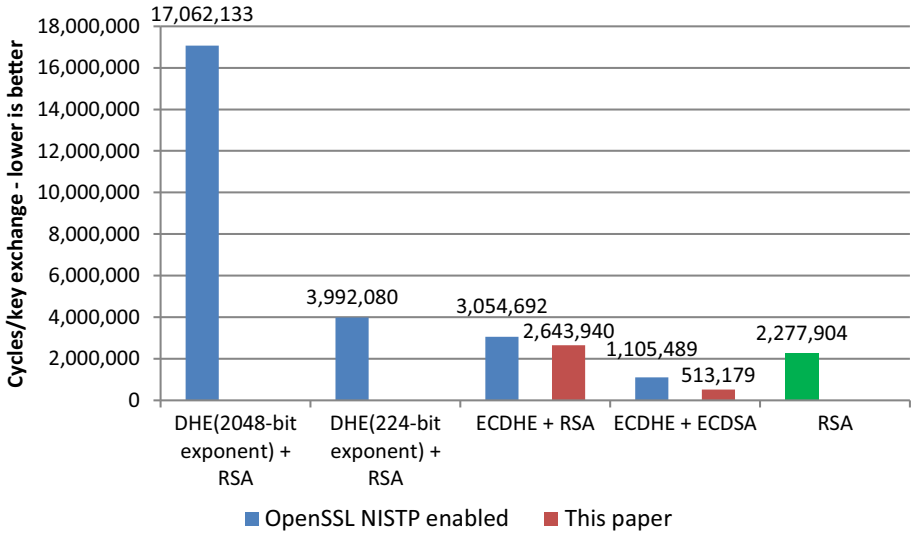

Fig. 10 The performance of several signature and key exchange algorithms that can be used for TLS handshake, and provide PFS. The presented results are obtained *after* the ECC optimizations have been applied, and are to be compared to the graph shown in Fig. 1

shows how the time spent on the different parts of the session (with and without the patch). The ECC computations are reduced from 45 % of the CPU time, to only 23 %.

8.2 Performance comparison of several public key handshakes

Figure 10 provides an update to Fig. 1. It compares the performance of several PFS combinations *after* our improved ECC implementation. From the figure we can see that the gap between using DHE+RSA vs. ECDHE+RSA grew significantly. In addition, using the combination of ECDHE+ECDSA is roughly five times faster than using RSA, while at the same time providing greater security (128 bit vs. 112 bit) and Perfect Forward Secrecy.

8.3 Performance estimation on future processors

A future Intel microarchitecture may introduce two new instructions, namely ADCX and ADOX that assist high-speed multi-precision integer arithmetic [12]. Together with the instruction MULX (which already exists in the Haswell microarchitecture), two "carry chains" can be parallelized, and this can lead to an efficient implementation of *MM*. We used these instructions to obtain a further optimized ECC implementation, and report our preliminary results.

Since there is not yet any processor with these instructions, the actual performance of the additional optimization cannot be measured at this point. Therefore, we took a different approach. The code can be compiled (using gcc version 4.8.0 and above), and executed on an emulator (Intel Software Development Emulator - SDE[4]). We used the SDE tool to count the number of executed instructions. Table 1 compares the instructions count of three implementations: OpenSSL and the two optimizations discussed here. The ADCX/ADOX-based implementation offers an incremental reduction of 22, 9 and 18 % in the instructions count for ECDH, ECDSA sign, and ECDSA verify, respectively (compared to our first optimization). The instructions are expected to have the same throughput and latency as the legacy ADC instruction. This gives a strong indication for a future performance improvement in the next generation's processor.

Table 1 The instructions count for three implementations (see explanation in the text)

	Instructions count		
	OpenSSL NISTP enabled	This paper, using ADD/ADC/MUL	This paper, using ADCX/ADOX/MULX
ECDH	2,338,113	972,520	757,114
ECDSA sign	882,879	412,982	375,773
ECDSA verify	2,336,523	1,077,205	880,185

It shows the potential improvement available by using the future ADCX/ADOX instructions

9 Discussion

Using a combination of algorithmic and software implementation improvements, we considerably sped up the ECC operations with the NIST P-256 curve. We also showed that this optimization provides a significant improvement at the server's performance level. This makes ECC with this NIST prime more attractive to use.

We also tested our software on earlier microarchitectures such as Intel® Penryn and Intel® Westmere. On both

[4] Available from http://software.intel.com/en-us/articles/intel-software-development-emulator.

```
p = 2^256-2^225+2^224-2^96+2^64-1
FFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFF
a = p-3
The base point G:
x_G =
0x32C4AE2C1F1981195F9904466A39C9948FE30BBFF2660BE1715A4589334C74C7
y_G =
0xBC3736A2F4F6779C59BDCEE36B692153D0A9877CC62A474002DF32E52139F0A0
the order of G, n =
0xFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFF7203DF6B21C6052B53BBF40939D54123
```

Fig. 11 The recommended parameters of the State Public Key Cryptographic Algorithm SM2 for public key [6], using a 256-bit elliptic-curve cryptography [5]. Note that is satisfies $a = p - 3$, and that the prime is an MF prime. This implies that all of the optimizations proposed here can be equally applied to this standard (with the appropriate pre-computed tables)

microarchitectures, our implementation gained significant speedup, compared to OpenSSL NISTP optimized implementation: 1.56×–1.58× were observed for sign, 1.41×–1.40× for verify, and 1.33×–1.36× for ECDH key computation.

Currently, the NIST P-256 is the only curve (with 128-bit security) that is used in TLS. However, other curves with comparable security exist, and there are attempts to adopt them as a standard as well. One such prominent curve is curve25519 [3]. We measured the performance over this curve (using the SUPERCOP performance benchmarking suite), and got the following results: generate key: 169,308 cycles, compute key: 161,188 cycles and sign at 60,780 cycles. The cost of a Diffie–Hellman (generate + compute) exchange with curve25519 is ∼330,000 cycles versus the approximate 365,000 cycles we achieved with our new patch (with P-256). This translates to a ∼10 % performance advantage for curve25519. The ECDHE + ECDSA performance of curve25519, gives ∼391,000 cycles on the server side, compared to about 510,000 of the P-256 (translating to a ∼30 % performance advantage to curve25519).

One of the properties of the NIST 256-bit prime, which we used for optimizing the underlying computations, was that p_{256} is an MF prime. However, this property is not unique for the NIST prime, and any ECC standard that uses an MF prime can enjoy the same optimization (of course, with the correct pre-computed tables being used, but this does not affect the resulting performance). Interestingly, there is already such a standardized case, namely the Chinese SM2 public key standard [6]. The relevant SM2 parameters are shown in Fig. 11.

Except for P-256, the only other NIST prime that is MF friendly is P-521. However, it has an optimized reduction routine that does not require Montgomery multiplications at all. P-224 is not MF, and P-384 is (only) 32-bit MF. Therefore, some of the optimizations described in this paper are not useful for these primes, on a 64-bit CPU. On the other hand, some of the optimizations (e.g., the use of larger tables, and AVX2 instructions) can be applied to any curve.

The implementation described in this paper can be seamlessly integrated into the OpenSSL library, as demonstrated in [10]. The code was contributed as a patch [10], for integration as a whole or in parts, for the benefit the open source community. It is currently in the process of integration into a future version of this library.[5]

We conclude with a word of caution with regard to intellectual properties and elliptic-curve cryptography. The information in this paper is provided as is. No license (expressed or implied) to any intellectual property right is granted by this paper. The authors and Intel assume no liability whatsoever and disclaim any expressed or implied warranty, including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright, or other intellectual property rights.

Appendix A

The configuration details of the platform that was used for the system profiling experiment:

CPU: Engineering sample of Intel microarchitecture codename Haswell, native speed of 2.9 GHz, 4 cores, 8 threads. For the experiment, we disabled 3 out of 4 cores and underclocked the CPU to 1 GHz, this is because otherwise, when using AES-GCM cipher, that performs at around 1 cycle/byte, the CPU outperforms the network card and idles most of the time.

Memory: 8GB DDR3 1,600 MHz, two-channel configuration.

Hard drive: Intel SSD X25-M 80GB.

[5] It is currently in the process of integration into a future version of this library (1.0.2), and can be found in the latest beta version (1.0.2 beta 3).

Software: Apache server version 2.4.4; OpenSSL development version (from September 9, 2013).

Network: Engineering sample of Intel 10Gbit Ethernet adapter.

References

1. Aciiçmez, O., Gueron, S., Seifert, J.P.: New branch prediction vulnerabilities in open SSL and necessary software countermeasures. In: Galbarith, S.D (ed.) Cryptography and Coding. LNCS, vol. 4887, pp. 185–203. Springer, Heidelberg (2007)
2. Barker, E., Roginsky, A.: Transitions: recommendation for transitioning the use of cryptographic algorithms and key lengths. NIST Special Publication 800–131A, NIST (2011). http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf
3. Bernstein, D.J.: Curve25519: new Diffie–Hellman speed records. In: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24–26, 2006. Proceedings, pp. 24–26. Springer, Heidleberg (2006)
4. Booth, A.D.: A signed binary multiplication technique. Q. J. Mech. Appl. Math. **4**(2), 236–240. Oxford University Press, Oxford (1951)
5. China Cryptography Administration: SM2 Elliptic curve recommended parameters (in Chinese). http://www.oscca.gov.cn/UpFile/2010122214836668.pdf
6. China Cryptography Administration: State Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves (2010, in Chinese). http://www.oscca.gov.cn/UpFile/2010122214822692.pdf
7. Dierks, T., Rescorla, E.: The transport layer security (TLS) protocol version 1.2. IETF RFC5246. IETF (2008). http://tools.ietf.org/html/rfc5246
8. Gueron, S., Krasnov, V.: [PATCH] Efficient and side channel analysis resistant 1024-bit and 2048-bit modular exponentiation, optimizing RSA, DSA and DH of compatible sizes, for AVX2 capable x86_64 platforms. OpenSSL patch (2013). http://rt.openssl.org/Ticket/Display.html?id=3054&user=guest&pass=guest
9. Gueron, S.: Efficient software implementations of modular exponentiation. J. Cryptograph. Eng. **2**, 31–43. Springer, Heidelberg (2012)
10. Gueron, S., Krasnov, V.: [PATCH] Fast and side channel protected implementation of the NIST P-256 elliptic curve, for x86–64 platforms. OpenSSL patch (2013). http://rt.openssl.org/Ticket/Display.html?id=3149&user=guest&pass=guest
11. Greenwald, G., MacAskill, E.: NSA prism program taps in to user data of Apple, Google and others. In: The Guardian (June 2013). http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data
12. Intel: Intel R architecture instruction set extensions programming reference (2013). http://download-software.intel.com/sites/default/files/319433-015.pdf
13. Käsper, E.: Fast Elliptic Curve Cryptography in OpenSSL. In: Danezis, G., Dietrich, S., Sako, K. (eds.) Financial Cryptography and Data Security. LNCS, vol. 7126, pp. 27–39. Springer, Heidelberg (2012)
14. Langley, A.: Protecting data for the long term with forward secrecy. In: Google Online Security Blog (2011). http://googleonlinesecurity.blogspot.co.il/2011/11/protecting-data-for-long-term-with.html
15. Montgomery, P.L.: Modular multiplication without trial division. Math. Comput. **44**, 519–521 (1985)
16. NIST: Mathematical routines for the NIST prime elliptic curves (2010). http://www.nsa.gov/ia/_files/nist-routines.pdf
17. NSS, Mozilla. https://developer.mozilla.org/en/docs/NSS
18. OpenSSL git repository. http://git.openssl.org/gitweb/
19. Renfro, S.: Secure browsing by default. In: Facebook, Facebook Engineering. https://www.facebook.com/notes/facebook-engineering/secure-browsing-by-default/10151590414803920
20. Satler, M., Housley, R.: Suite B Profile for Transport Layer Security (TLS). IETF RFC6460. IETF (2012). http://tools.ietf.org/html/rfc6460
21. Solinas, J.A.: Generalized Mersenne numbers. Center for Applied Cryptographic Research. University of Waterloo,Technical Report (1999)