# Speeding-Up P-256 ECDSA Verification on x86-64 Servers

Nir Drucker and Shay Gueron

**Abstract—**ECDSA is a frequently used signature scheme that has attracted a great deal of software and hardware optimization efforts. In particular, the NIST P-256 curve is currently used for most of the TLS communication worldwide. This paper proposes some observations that lead to additional optimizations. The ECDSA verification includes two main bottlenecks: a) modular inversion (modulo the group order); b) two scalar-point multiplications on the underlying elliptic curve (the points are the group's generator and the signer's public key). One observation is that when one point is multiplied by more than one scalar, the multiplications can be accelerated by precomputing some intermediate values. For example, OpenSSL stores a hard-coded multiplication table for the generator of the NIST P-256 curve. Another observation leveraged here is the fact that ECDSA verification uses only public information, and therefore side channel mitigation techniques are not necessary. We show how these observations lead to a speedup of 3.4× over the NIST P-256 ECDSA verification of OpenSSL [1], using $\sim 150$ Kb of additional memory space. Some of the optimizations that are offered here, have been recently integrated into BoringSSL [2], [3] achieving a $1.15\times$ speedup over its previous implementation.

**Index Terms—**ECDSA, signature verification, software optimization, BoringSSL, OpenSSL

## 1 Motivation

SERVERS in a cloud environment often communicate with a fixed set of servers and clients and in such cases the same public/private key pair is used for signing and verifying multiple signatures. Optimization for such cases has been mentioned but surprisingly major libraries (e.g., OpenSSL [1]) do not support it. In this note we demonstrate how these optimizations combined with a few other techniques can be patched into OpenSSL and speedup ECDSA verification by $3.4\times$.

The techniques are described in Section 3, concrete implementation details are given in Section 4, and the results are reported in Section 5.

## 2 Preliminaries and Notation

A uniform random selection of $x$ from $[a, b] = \{x \mid a \leq x \leq b, x \in \mathbb{Z}\}$ is denoted by $x \xleftarrow{\$} [a, b]$. Scalar point multiplication is denoted by $s \times R$, where $s$ is a scalar and $R$ is a point. A polynomial $a_l \cdot 2^l + \ldots + a_0 \cdot 2^0$ of degree $l$ and coefficients $a_i \in \{0, 1\}$, $0 \leq i \leq l$ is represented as an $l$-bit array $A = a_l \ldots a_0$ ($A[i] = a_i$). For an array of bits $A$, the function $trunc_{256}(A)$ returns the string $a_{255} \ldots a_0$ (in case $l < 256$ it sets $a_{l+1} = \cdots = a_{255} = 0$). Let $\gg s$ denote the right shift operation by $s$ bits. Numbers written in hexadecimal base are denoted with a prefix `0x` (e. g., `0x1F` is 31 decimal). Hereafter, $\mathcal{H}$ refers to a hash function (in our context SHA256).

*ECDSA.* ECDSA signature/verification assumes that the signer and verifier have agreed on a curve. In this note, the agreed curve is NIST P-256 curve [4] (and we denote by $P$ its prime field over which it is defined), the generator is $G$ and the generator's order of $G$ is $N$ (i. e., $N \times G$ is the point at infinity $O$). The ECDSA signing and verification protocols are outlined in Algorithm 1.

**Algorithm 1.** ECDSA Signature Protocols [4]

**Input:** $G$ (a group generator), $N$ (the order of $G$), $d$ (a private key), $Q$ (the associated public key), $m$ (a byte array)
**Output:** $(r, s)$, where $r, s \in [1, N-1]$.

1: **procedure** ECDSASIGN
2: $e = trunc_{256}(\mathcal{H}(m))$
3: $k \xleftarrow{\$} [1, N-1]$
4: $(x, y) = k \times G$
5: $r = x \pmod N$
6: **if** $r = 0$ **then** goto step 3
7: $s = k^{-1}(e + rd) \pmod N$
8: **if** $s = 0$ **then** goto step 3
9: **return** $(r, s)$

10: **procedure** ECDSAVERIFY
11: **if** $r \notin [1, N-1]$ or $s \notin [1, N-1]$ **then return** False
12: $e = trunc_{256}(\mathcal{H}(m))$
13: $w = s^{-1} \pmod N$
14: $u_1 = ew \pmod N$
15: $u_2 = rw \pmod N$
16: $(x, y) = u_1 \times G + u_2 \times Q$
17: **if** $r = x \pmod N$ **then return** True
18: **else return** False

*Elements of ECDSA Software Implementation.* The (relevant parts of the) Elliptic Curve Cryptography (ECC) code implementations in OpenSSL [1] and BoringSSL [5] consist of almost the same code flows and APIs. Hereafter, code objects and functions are denoted in `pcr` letters (and we adopt names used in OpenSSL). A detailed list is given in Appendix A. The code defines three principal objects:

- `EC_GROUP` - holds data on a specific elliptic curve group (e.g., NIST P-256)
- `EC_KEY` - holds the private and public keys associated with a specific (pre-determined) `EC_GROUP`
- `EC_POINT` - holds a specific point on a (pre-determined) curve

The implementations use Jacobian representation (see [6]), and transform the inputs (given in affine coordinates) to Jacobian coordinates and eventually transform the results back to affine representation.

## 3 Optimization Techniques

We identify three principal optimizations:

- Performing the final verification comparison in Jacobian's coordinates (instead of in affine coordinates).
- Modifying the ECDSAVerify function to a variable time implementation (dropping the unnecessary constant-time side channel protection that is needed only for signing).
- Precomputing a table with some multiplies of the public key to speed-up point multiplication.

The details follow:

### 3.1 Optimizing the Final Comparison

Step 17 of Algorithm 1 is implemented in the `ECDSA_verify` function by comparing coordinates in affine representation (comparing the $x$ coordinate with the signature $r$). Converting $x$ from the Jacobian's representation to the affine coordinate involves

• *The authors are with the University of Haifa, Haifa 3498838, Israel, and also with Amazon, Seattle, WA 20189. E-mail: drucker.nir@gmail.com, shay@math.haifa.ac.il.*

*Manuscript received 7 Feb. 2019; revised 11 Mar. 2019; accepted 31 Mar. 2019. Date of publication 14 Apr. 2019; date of current version 9 May 2019.*
*(Corresponding author: Nir Drucker.)*
*Recommended for acceptance by J. Hughes.*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/LOCS.2019.2911063*

modular inversion (namely, $x^{affine} = xz^{-2} \pmod P$). Instead, we convert $r$ to its Jacobian representation (by, $r_{Jacobian} = rz^2 \pmod P$) where the cost is only modular multiplication.

In this context, note that Step 5 of ECDSASign reduces $x$ modulo $N$ (i.e., $x < N$). On the other hand, $x$ (computed in Step 16 of ECDSAVerify) is reduced only modulo $P$ ($P > N$). Therefore, the verification needs to check whether (or not) $x = r$ or $x = r + N$. Note that $P > N$ is a property of curve P-256. Other curves may have different properties.

### 3.2 Pre-Computing Multiples of a Point

If a public key $Q$ is used more than once, it pays to pre-compute some multiplication table of $Q$, store it and subsequently use it. This technique is used for the generator point as described in [7] (the related optimized code was contributed to OpenSSL [8]). This contributed implementation uses the windowing method and Booth encoding [9] with a window of size 7. Consequently, there are $\lceil 256/7 \rceil = 37$ windows, and hence $37 \times 2$ tables to store (in either affine/Jacobian coordinates). The table needs to store the values ($i \in [0, 36]$, $j \in [0, 2^6 - 1]$[1])

$$Table[i][j] = (2^{7i} \cdot j) \times Q \pmod P, \quad (1)$$

its size is $(256/8) \times 37 \times 2 \times 2^6 = 151,552$ bytes ($\sim 148$ KB). A Sagemath script for generating this table is given in Appendix B.

## 4 Putting It All Together

We implemented the proposed optimizations as C, x86 Assembly, and Perl code patches for OpenSSL [10] and BoringSSL [2], [3] ready for performance profiling. We used x86 platforms for demonstrating the performance advantage of these optimizations. Note that code paths for other platforms can also adopt these optimizations. We measured performance on three different platforms, all running Linux (Ubuntu 16.04 LTS), characterized as follows:

- The "SKX server" platform. A `c5.9xlarge` AWS EC2 instance equipped with the latest 6th Generation Intel Core$Y$ processor ("SkyLake" [SKX]) - Intel Xeon Platinum 8124M CPU at 3.00 GHz Core i$5-750$. This platform has 70 GB RAM, 32K L1d and L1i cache, 1,024K L2 cache, and 25,344K L3 cache. It was configured to disable the Intel Turbo Boost Technology from the OS (by setting bit 38 of MSR `0x1A0`).
- The "KBL desktop" platform. An Intel desktop with the 7th Intel Core$^{TM}$ Generation (Micro-architecture Codename "Kaby Lake" [KBL]) running at 3.60 GHz Core$^{TM}$ i$7-7700$. This platform has 16 GB RAM, 32K L1d and L1i cache, 256K L2 cache, and 8,192K L3 cache, where the Intel Turbo Boost Technology was turned off and the Enhanced Intel Speedstep Technology was disabled.
- The "IVB desktop" platform. An Intel desktop with the 3th Intel Core$^{TM}$ Generation (Micro-architecture Codename "Ivy Bridge" [IVB]) running at 2.90 GHz Core$^{TM}$ i7-3520$M$. This platform has 8 GB RAM, 32K L1d and L1i cache, 256K L2 cache, and 4,096K L3 cache.

The results were obtained by running the built-in utility "speed" of OpenSSL/BoringSSL.

### 4.1 Variable Time Implementation

The verification in the `ossl_ecdsa_verify_sig` function does not handle secret data, therefore the channel mitigation that makes ECDSASign side channel protected is not required here. Accordingly, we optimized a dedicated code path for ECDSAVerify with variable time implementation. To this end, we replaced the call to `EC_POINT_mul` with `EC_POINT_vartime_mul` and the call to `ec_group_do_inverse_ord` with `ec_group_do_inverse_ord_vartime`. The various functions `ecp_nistz256_points*_mul` differ in four locations:

(1) The function `ecp_nistz256_points_mul` accesses the pre-computed table by invoking a constant time x86 assembly function (`ecp_nistz256_gather_w7`). Our functions uses `memcpy` directly.

(2) The pre-computed table is designed for Booth encoding. During multiplication, this encoding involves points' negation. The function `ecp_nistz256_points_mul` performs a constant time negation by

```
ecp_nistz256_neg(tmp, point.Y);
copy_conditional(point.Y, tmp window_value & 1);
```

we replaced this lines with (affording an if statement):

```
if((window_value & 1) == 1) {
    ecp_nistz256_neg(point.Y, point.Y);
}
```

(3) Our implementation skips point addition when `window_value` $= 0$. A constant time implementation must perform the same operation regardless of the value of `window_value`.

(4) Our function `EC_POINT_vartime_mul` receives $Q$ and $G$ as inputs and uses an additional pre-computed multiplication table for $Q$. By contrast, the function `EC_POINT_mul` (used for signing) that also receives multiple points as inputs, does not use fixed points except for $G$. Consequently, it cannot gain from pre-computed tables.

#### 4.1.1 Variable Time Modular Inversion

We replaced the function `ecp_nistz256_inv_mod_ord` with `ecp_nistz256_inv_mod_ord_vartime`. This is an internal function that serves as a wrapper of a new x86 assembly function that implements the Binary Extended Euclidean Algorithm for calculating the modular inverse.

### 4.2 Storing the Pre-Computed Multiplication Table

A memory pointer to $Q$'s pre-computed multiplication table can be defined as a structure member of either the `EC_KEY` object or the `EC_POINT` object. We allocate it as a member of `EC_POINT` because the table is used for point multiplication (in particular when the function `EC_POINTs_mul` is called). This decision makes the code ready for possible tables for other points. In contrast, OpenSSL stores a pointer to the pre-computed multiplication table (of $G$) as a member (`pre_comp`) of `EC_GROUP`. To resolve this incompatibility we removed the `pre_comp` member of the `EC_GROUP` object and instead use our new `EC_POINT`'s `pre_comp` member. This new (pointer) member of the `EC_POINT` object requires only slightly more memory than before because our code populates the table only upon request. The code of the function `ecp_nistz256_mult_precompute` was reused for populating this table (technically, we exported the "generic" table population code to a new function `ecp_nistz256_precompute_mult_for_point`) while leaving the checks that are related to $G$ in the original function. We also added a function callback to the `ec_method_st` structure, that we call `precompute_mult_for_point`. In order to avoid a new APIs we kept this new interface internal. Subsequently, we modified `EC_KEY_precompute_mult` (that used to call only the `EC_GROUP_precompute_mult` API) to also call our new API on the public key (if set). This supports the user's decision on using the pre-compute multiplication table.

1. We use $2^6$ instead of $2^7$ due to the Booth encoding. In addition, the case $j = 0$ can also be optimized. See explanations in [7].

TABLE 1
ECDSA Verify OpenSSL Patch [10]: Achieved Speedup (without the Final Inversion Optimization)

| Platform | Baseline (verify/sec) | Patched (this work) (verify/sec) | Speedup factor |
|---|---|---|---|
| IVB | 7,835 | 26,983 | 3.44 |
| KBL | 15,630 | 50,609 | 3.24 |
| SKX | 14,879 | 46,857 | 3.15 |

TABLE 2
Speed Up of Our BoringSSL ECDSA Verify Patch [2] (Without the Pre-Computed Tables Optimization)

| Platform | Baseline (verify/sec) | Patched (this work) (verify/sec) | Speedup factor |
|---|---|---|---|
| KBL | 16,171 | 19,254 | 1.19 |
| SKX | 14,904 | 17,619 | 1.18 |

**Remark 1.** OpenSSL stores a hard coded copy of the multiplication table for NIST P-256's generator in `ecp_nistz256_table.c`. In case the user does not call `EC_KEY_precompute_mult`, OpenSSL will still use the stored table during multiplication. Thus, calling this API is really needed only when the user wants to pre-compute a multiplication table for the public key. Calling this API with other curves will become slower (because two tables are being calculated).

**Remark 2.** By our definition, `EC_POINT_precompute_mult` needs only a point as its input (that includes a pointer to a group object). However, modifying the API will break compatibility with OpenSSL. We chose not to do that change and call our internal function only from within the `EC_KEY_precompute_mult` API.

## 5 Results

This section reports the performance improvements achieved on different platforms, by patching different libraries, and different combinations of optimization methods. All the measurements are sampled by using the built in "speed" utility (of OpenSSL/BoringSSL).

Table 1 shows the speedup (on different platforms) achieved by applying our OpenSSL patch over the original library. This patch includes optimization 2,3 but does not include the final comparison optimization. For example, we observe a 3.44x speedup on IVB desktop platform. Note that the patch enjoys higher speed up on the "older" CPUs. This is because OpenSSL (as well as BoringSSL) includes optimizations that leverage new features of the "newer" CPUs, and this improves the baseline performance.

Table 2 shows the improvement achieved by applying our BoringSSL patch [2] over the original library. This patch implements optimization 1,2 but does not include the pre-computed tables optimization.

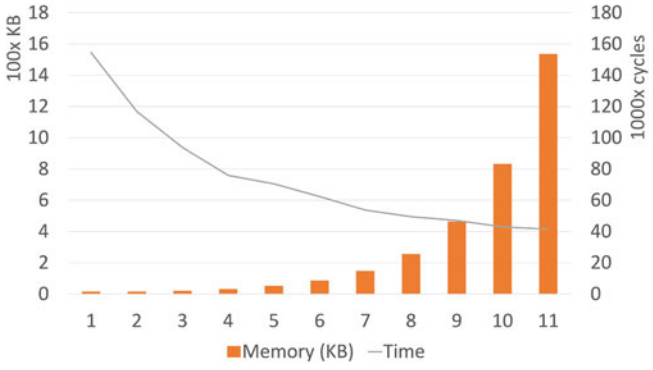


Fig. 1. Different window sizes (in the range $1, \ldots, 11$) lead to different trade-offs between memory costs and performance gains.

Fig. 1 presents the memory-performance trade-off obtained by using different window sizes (from 1 to 11) for the pre-computed tables. The optimal balance seems to be attained near window size 6. Above that window size, the memory costs grow faster than the incremental performance gains.

TABLE 3
APIs Used by OpenSSL to Perform Multiplication

| API | Description |
|---|---|
| `EC_POINTs_mul` | Calculates the value $s_0 \times G + s_1 \times q_1 + \ldots + s_n \times q_n$ where $n \in \mathbb{Z}_{\geq 0}$, and for every $i \in [1, n]$: $s_i \in \{0, 1\}^{256}$ and $q_i$ is a curve point. |
| `EC_POINT_mul` | Same as `EC_POINTs_mul`, where $n \leq 1$. |
| `EC_GROUP_ precompute_mult` | Stores a table with some multiples of $G$. This will be used later for accelerating the `EC_POINT_mul` function. |
| `EC_KEY_ precompute_mult` | Calls `EC_GROUP_precompute_mult` on the `EC_GROUP` member of the referred `EC_KEY`. |
| `EC_KEY_generate_key` | Generates and stores a new set of keys (private/public). |
| `EC_KEY_set_public_key` | Sets the (`EC_POINT`) public key of the associated `EC_KEY`. |

## Appendix A
## OpenSSL Code

Table 3 presents a partial list of OpenSSL APIs that use these objects.

The `crypto/ec/ecdsa_ossl.c` file contains the ECDSA signing (`ossl_ecdsa_sign_sig`) and verification (`ossl_ecdsa_verify_sig`) functions. The file `crypto/ec/ecp_nistz256.c` contains an optimized implementation dedicated for the NIST P-256 curve. It includes two main functions `ecp_nistz256_mult_precompute` and `ecp_nistz256_points_mul` that implement the `EC_GROUP_mult_precompute` and `EC_POINTs_mul` APIs, respectively.

## Appendix B
## Generating a Multiplication Table

The following Sagemath sample code demonstrates how to generate the multiplication table of a specific public key point in a format that can directly be included by OpenSSL. We used backslash to brake long lines (and numbers), this is not part of the Sagemath language.

```
def print_header():
    print "#if defined(__GNUC__)"
    print "__attribute((aligned(4096)))"
    print "#elif defined(_MSC_VER)"
    print "__declspec(align(4096))"
    print "#elif defined(__SUNPRO_C)"
    print "# pragma align 4096(ecp_nistz256_precomputed)"
    print "#endif"
    print "static const BN_ULONG",
    print "ecp_nistz256_precomputed[37][64 *",
    print "sizeof(P256_POINT_AFFINE) /",
    print " sizeof(BN_ULONG)] = {"
    print "    {"
```

**Listing 1.** The print_header procedure, prints the table declaration.

```
def print_val(x, last_val):
    for i in range(4):
        if (i % 2 == 0):
            print "     ",
        m_lo = x % (2^32)
        x = x >> 32
        m_hi = x % (2^32)
        x = x >> 32
        print ("TOBN(0x%0.8x, 0x%0.8x)" % (m_hi, m_lo)),
        if (i % 2 == 0):
            print(","),
        elif (i == 1):
            print(",")
        elif ((last_val != True) and (i == 3)):
            print(",")
        else:
            print
```

**Listing 2.** The print_val procedure, prints a 256-bit point coordinate (x/y) in the format required by OpenSSL. The last val boolean flag indicates whether this is the last value in a row of the table.

```
def to_mont(x, p):
    r = 2^(256)
    return ((x * r) % p)
def print_table(P, prime):
    print_header()
    for i in range(37):
        for j in range(1,64+1):
            t = (2^(7*i)) * j
            P1 = t * P
            print_val(to_mont(Integer(P1[0]), prime), False)
            last_val = ((i!=36) and (j == 64))
            print_val(to_mont(Integer(P1[1]), prime), last_val)
        if (i < 36):
            print "    },{"
        else:
            print "    }};"
```

**Listing 3.** The main procedure, prints the entire table.

```
p256 = (2^256) - (2^224) + (2^192) + (2^96) - 1
a256 = p256 - 3
b256 = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b
FF = GF(p256)
EC = EllipticCurve([FF(a256), FF(b256)])

# Use default generator
gx = Integer(0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296)
gy = Integer(0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5)
G = EC(FF(gx), FF(gy))

# Curve ord:
o = 115792089210356248762697446949407573529996955224135760342422259061068512044369
EC.set_order(o)
```

**Listing 4.** These lines define the parameters of the NIST P-256 curve.

```
sx = Integer(0x7CF27B188D034F7E8A52380304B51AC3C08969E277F21B35A60B48FC47669978)
sy = Integer(0x07775510DB8ED040293D9AC69F7430DBBA7DADE63CE982299E04B79D227873D1)
S = EC(FF(sx), FF(sy))
print_file(S, p256)
```

**Listing 5.** Usage example: the public key associated with the private key d = 2.

```
#if defined(__GNUC__)
__attribute((aligned(4096)))
#elif defined(_MSC_VER)
__declspec(align(4096))
#elif defined(__SUNPRO_C)
# pragma align 4096(ecp_nistz256_precomputed)
#endif
static const BN_ULONG ecp_nistz256_precomputed[37][64 *
        sizeof(P256_POINT_AFFINE) / sizeof(BN_ULONG)] = {
 {
  TOBN(0x850046d4, 0x10ddd64d), TOBN(0xaa6ae3c1, 0xa433827d),
  TOBN(0x73220503, 0x8d1490d9), TOBN(0xf6bb32e4, 0x3dcf3a3b),
  TOBN(0x2f3648d3, 0x61bee1a5), TOBN(0x152cd7cb, 0xeb236ff8),
  TOBN(0x19a8fb0e, 0x92042dbe), TOBN(0x78c57751, 0x0a5b8a3b),
  TOBN(0x74b0b50d, 0x46918dcc), TOBN(0x4650a6ed, 0xc623c173),
      ...
```

**Listing 6.** Output example.

## ACKNOWLEDGMENTS

We thank Matthew J. Campagna for his valuable feedback on the optimizations. We also thank the BoringSSL team for reviewing, handling, and organizing our code contribution according to the BoringSSL coding style, finally for integrating the patch into BoringSSL. This research was supported by: the Israel Science Foundation (grant No. 1018/16); the Center for Cyber Law & Policy at the University of Haifa, in conjunction with the Israel National Cyber Directorate in the Prime Ministers Office.

## REFERENCES

[1] OpenSSL, "OpenSSL commit c36b39b5cd685fc5eae84ece247e7873a27d8834," Jul. 2018. [Online]. Available: https://github.com/openssl/openssl/commit/c36b39b5cd685fc5eae84ece247e7873a27d8834

[2] N. Drucker and S. Gueron, "Speed up ECDSA verify on x86–64," Nov. 2018. [Online]. Available: https://boringssl-review.googlesource.com/c/boringssl/+/31304

[3] D. Benjamin, "Revert "Revert "Speed up ECDSA verify on x86–64.""," Nov. 2018. [Online]. Available: https://boringssl-review.googlesource.com/c/boringssl/+/32924

[4] Fips PUB, "186–2. digital signature standard (DSS)," *Nat. Inst. Standards Technol.*, vol. 20, 2000, Art. no. 13.

[5] BoringSSL, "BoringSSL commit 7f7e5e231efec6e86d6c7d3fd1b759be1cece156," Nov. 2018. [Online]. Available: https://boringssl.googlesource.com/boringssl/+/7f7e5e231efec6e86d6c7d3fd1b759be1cece156

[6] D. Bernstein and T. Lange, "Explicit-formulas database," 2007. [Online]. Available: http://www.hyperelliptic.org/EFD/oldefd/jacobian.html

[7] S. Gueron and V. Krasnov, "Fast prime field elliptic-curve cryptography with 256-bit primes," *J. Cryptographic Eng.*, vol. 5, no. 2, pp. 141–151, Jun. 2015. [Online]. Available: https://doi.org/10.1007/s13389-014-0090-x

[8] S. Gueron and V. Krasnov, "Fast and side channel protected implementation of the NIST P-256 Elliptic Curve, for x86-64 platforms," 2013. [Online]. Available: https://groups.google.com/forum/#!topic/mailing.openssl.dev/q6qitLHf874

[9] A. D. Booth, "A signed binary multiplication technique," *The Quart. J. Mech. Appl. Math.*, vol. 4, no. 2, pp. 236–240, 1951.

[10] N. Drucker and S. Gueron, "OpenSSL patch demonstration," 2019. [Online]. Available: https://github.com/drucker-nir/openssl/tree/FastECDSA-x86