Efficient and Secure Elliptic Curve Cryptography Implementation of Curve *P-256*

Mehmet Adalier[1]
Antara Teknik, LLC

Abstract

Public key cryptography has become the *de facto* standard for secure communications over the Internet and other communications media such as cellular and Wi-Fi. Elliptic curves offer both better performance and higher security than first generation public key techniques and are gaining acceptance as the foundation for future Internet security such as the security-enhanced Border Gateway Protocol (BGPSEC). In this paper, we present a performance optimized and side-channel-attack resistant implementation of the NIST Curve *P-256* which provides 128-bits of security. We also discuss operation time vs. storage trade-offs for various approaches.

Introduction

The reliable functioning of critical infrastructure, such as the Internet, is imperative to the national and economic security of United States [1] especially as the frequency and complexity of cyber-security threats are increasing significantly. The currently deployed Border Gateway Protocol (BGP), which was last updated in 2006 [2,3], does not include provisions for security features and is vulnerable to malicious attacks targeting the control plane. These attacks can be perpetuated in a number of ways [4,5,6] and could cause significant failures and instability. Moreover, perpetuators can deny service, re-route traffic to malicious hosts, and expose network topologies. There have been significant efforts over the years to add robustness to BGP and to provide Best Common Practice (BCP) guidance for the same [7,8,9].

The Internet Engineering Taskforce (IETF) is currently developing BGPSEC (BGP with Security) [10], an extension to BGP with the intention to provide path security for BGP route advertisements. The extension is meant to provide resiliency against route hijacks and Autonomous System (AS) path modifications. Specifically, two mechanisms: i) route-origin validation [11]; and ii) path validation are being defined [10]. As described in RFC 6480 [12] the Resource Public Key Infrastructure (RPKI) provides the initial step used to validate BGP routing data. First, holders of AS number and IP address resources are issued RPKI Resource Certificates, which establish a binding between them and cryptographic keys for digital signature verification. Furthermore, a Route Origination Authorization (ROA), which is a digitally signed object, allows holders of IP address resources to authorize specific ASes to originate routes. BGP speakers can use ROAs to ensure that the AS which originated the received route, was in fact authorized to originate that route.

ECDSA *P-256*, a prime curve that has been used extensively in critical infrastructure projects, is being used as the Elliptical Curve Digital Signature Algorithm for AS-path signing and verification in the BGPSEC protocol [10]. The performance efficiency of ECDSA *P-256* is imperative to meet strict Internet routing table convergence requirements [13]. Thus the viability of BGPSEC adoption is dependent on the availability of high performance implementations of ECDSA *P-256*.

In this paper we discuss key implementation areas and optimization opportunities, and show that it is possible to implement ultra fast and secure ECDSA for the curve *P-256*, delivering full 128-bits of security, on low-cost and low-power commercially available hardware. Furthermore, our work can be extended to optimize other prime curves such as Curve *P-521*, which provides 256-bits of security.

1.0 ECDSA Overview

Elliptical Curve Cryptology has been extensively studied and documented [14,15]. This paper is focused on applied cryptography and implementation aspects rather than mathematical proofs of underlying theorems. This section provides a brief overview of the fundamentals.

[1] This material is based upon work supported by the National Institute of Standards and Technology (NIST) under cooperative agreement 70NANB14H289. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of NIST.

1.1 ECDSA Parameters

For proper implementation of ECDSA the use of a specific set of elliptic curve domain parameters are required for digital signature generation and verification. These domain parameters may be used for extended time periods (i.e. over multiple sessions). Specifically the applicable ECDSA Domain Parameters are:

q, the size of the underlying field
a, elliptic curve parameter (equal to q-3 for *P-256*)
b, elliptic curve parameter
$G = (x_G, y_G)$, a point on the curve, known as the base point,
n, the order of the base point G.

The equation of the curve is generally given as

$$y^2 = x^3 + ax + b \bmod q$$

For NIST Prime Curves which include *P-256*, $a = q - 3$, and with this value of a, the equation is equivalent to the one given in FIPS 186-4 [16], namely:

$$y^2 = x^3 - 3x + b \bmod q$$

1.2 ECDSA Signature Generation

The inputs to ECDSA signature generation are: i) a message, M; ii) the appropriate curve domain parameters; iii) the appropriate Hash function [17]; and iv) the private key d. The output of the process is a pair of integers (r, s), each in the interval $[1, n - 1]$. The process is defined as [18,19]:

1. Generate (k, k^{-1}), where k is the per message secret number and k^{-1} is its inverse modulo n
2. Compute the elliptic curve point $R = kG = (x_R, y_R)$
3. Compute $r = x_R \bmod n$
4. Compute $H = \text{Hash}(M)$
5. Convert the bit string H to an integer e : $e = \Sigma^{H}_{(i=1)} 2^{H-i} * b_i$, where $b_1, b_2, ..., b_H$, is the bit string to be converted
6. Compute $s = (k^{-1} * (e + d * r)) \bmod n$
7. Return (r, s)

1.3 ECDSA Signature Verification

The inputs to ECDSA signature verification are: i) the received message M'; ii) (r', s'): the received signature on M'; iii) the appropriate curve domain parameters; iv) the appropriate Hash function; and iv) the public key Q. The output of the process is an indication of whether the supplied signature is valid or not. The process is defined as [18,19]:

1. If r' and s' are not both integers in the interval $[1, n - 1]$, output INVALID
2. Compute $H' = \text{Hash}(M')$
3. Convert the bit string H' to an integer e' by using: $e' = \Sigma^{H'}_{(i=1)} 2^{H'-i} * b_i$, where $b_1, b_2, ..., b_{H'}$, is the bit string to be converted
4. Compute $w = (s')^{-1} \bmod n$
5. Compute $u_1 = (e' * w) \bmod n$ and $u_2 = (r' * w) \bmod n$
6. Compute the elliptic curve point $R = (x_R, y_R) = u_1G + u_2Q$
7. Compute $v = x_R \bmod n$
8. Compare v and r'. If $v = r'$, output VALID; otherwise, output INVALID.

Note that domain parameters, k and d for *P-256* are 32-Bytes long each where as the points on the curve such as G and Q (public key) consist of 32-Byte x- and 32-Byte y-values each. The total length of the signature generated is 64 bytes (r 32 bytes, s 32 bytes). Given that most modern compute engine (e.g. CPUs) registers are either 32 or 64 bits, ECC arithmetic operations are performed by using multi-precision arithmetic, which require significant compute cycles for basic mp-integer operations (i.e. field operations) such as multiply, invert, and mod.

2.0 Elliptic Curve Point Representation and Group Level Operations

Assume E to be an elliptic curve over a prime field F_p with the affine equation $y^2 = x^3 - 3x + b$. Defining two points on the curve as $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ with $P_1 \neq -P_2$, then $P_3 = P_1 + P_2 = (x_3, y_3)$ is [20]:

$x_3 = \lambda^2 - x_1 - x_2$, and $y_3 = \lambda(x_1 - x_3) - y_1$, and

$\lambda = (y_2 - y_1)/(x_2 - x_1)$ when $P_1 \neq P_2$, and

$\lambda = (3x_1^2 - 3)/(2y_1)$ when $P_1 = P_2$

Since general addition only works when $P_1 \neq P_2$, addition for the case $P_1 = P_2$ is referred as point doubling. Prime field inversions are considerably more expensive in compute resource requirements than field multiplications. Thus representing points using projective coordinates may be beneficial. Using Jacobian projective coordinates [21], it can be shown that the projective point $(X : Y : Z)$, where $Z \neq 0$, corresponds to the affine point $(X/Z^2, Y/Z^3)$, and to the point at infinity (i.e. the identity element) when $Z = 0$.

Formulas for addition in mixed Jacobian-Affine coordinates are given as:

$(X_3 : Y_3 : Z_3) = (X_1 : Y_1 : Z_1) + (X_2 : Y_2 : 1)$, where

$A = X_2 . Z_1^2$, $B = Y_2.Z_1^3$,

$C = A - X_1$, $D = B - Y_1$,

$X_3 = D^2 - (C^3 + 2X_1.C^2)$;

$Y_3 = D. (X_1.C^2 - X_3) - Y_1.C^3$;

$Z_3 = Z_1.C$

Formulas for doubling in Jacobian coordinates are:

$(X_3 : Y_3 : Z_3) = 2(X_1 : Y_1 : Z_1)$, where

$A = 4X_1.Y_1^2$, $B = 8Y_1^4$,

$C = 3(X_1 - Z_1^2) . (X_1 + Z_1^2)$, $D = -2A + C^2$,

$X_3 = D$;

$Y_3 = C. (A - D) - B$;

$Z_3 = 2Y_1 . Z_1$,

where the operations are performed using field arithmetic with multi-precision positive integers.

As previously discussed, the ECDSA Sign Operation requires the multiplication of the curve base point G (x_G, y_G), with a scalar k, which is also referred as the One Time Secret Number (OTSN). For example, for curve *P-256*, k is a 256-bit Deterministic Random Number with security strength of at least 128 bits. The scalar multiplication is quite compute intensive and dominates the execution time of elliptic curve cryptographic operations. Considerable effort has been spent on minimizing the scalar multiplication time [22,23,24]. Assuming all computations are actually carried out, as in the basic Right to Left Binary Method shown as Algorithm 1, the expected running time is approximately $m/2$ point additions and m point doublings (denoted $0.5mA + mD$), where m is the length of the binary number k [expected number of ones in binary k is about $m/2$]. For *P-256* this equates to $0.5(256)A+256D = 128A + 256D$ which requires a considerable amount of compute time.

INPUT: $k = (k_{t-1}, \ldots, k_1, k_0)_2$, $P \in E(F_q)$
OUTPUT: $Q = kP$

1. $Q \leftarrow \infty$
2. For i from 0 to $t-1$ do
   2.1 If $k_i = 1$ then $Q \leftarrow Q + P$
   2.2 $P \leftarrow 2P$
3. Return (Q)

Algorithm 1) Right to Left Binary Method for Point Multiplication

3.0 Side-channel Attack Considerations

Side-channel attacks on implementations of cryptosystems may include timing or power consumption measurements in order to reveal secret information such as the OTSN or the private key. In elliptic curve cryptosystems, implementations of point multiplication algorithms are the primary targets for side-channel attacks [25,26,27]. Straightforward implementations of elliptic curve point multiplications, such as Algorithm 1 are exceptionally vulnerable to simple SCA since they employ both point addition and point doubling. Given the fact that point adding and doubling require substantially different formulas, bits of the OTSN could be extracted in a power consumption trace if the double-and-add algorithm is used for point multiplication. Differential power analysis is an attack [28] that enables extraction of a secret key stored in a cryptographic system where an adversary monitors the power consumption of the cryptographic system and then statistically analyzes the collected power signal data in order to extract the secret key.

Data randomizing is a well-known DPA countermeasure, by which the intermediate data may be randomly transformed inside the cryptographic system. The technique mitigates leakage that can be used by a DPA since the intermediate data is no longer predictable. Additionally, Liardet and Smart [29] and Joye and Quisquater [30] have proposed reducing information leakage by using special representations of points.

There are several approaches to reducing the running time of the scalar multiplication algorithm. One is using alternative representations of k in order to reduce the number of one-bits (and hence reduce number of point additions) such as NAF [31] along with pre-computation of point doublings (if the point is known). While alternative representations of k can introduce a performance benefit, the re-coding that is often needed may be susceptible to Side Channel Attacks because an adversary could use DPA to reveal portions of the secret information [32].

4.0 Optimization Methodology

There has been considerable research conducted to increase the security and the performance of ECC algorithms. Side-channel-attack resilience needs to be inherently built into core functions where applicable in an optimized fashion, rather than included as an after-thought. Performance can be increased via algorithmic or mathematical methods as well as with the facilitation of target platform features with low-level implementation techniques. While CPU core frequencies have stabilized in the range of 3 to 4HGz

with minimal potential for substantial increases, new instructions and platform features can often improve the performance of algorithms or methods which were thought to be too slow even only a few years ago. For example, the latest Intel® Architecture processors support large Last Level Caches, fast memory access, and new 64-bit integer arithmetic instructions. Furthermore, even the low-power embedded CPUs provide multiple 64-bit cores, with the flagship CPUs providing up to 18 cores. In this paper, we concentrate our discussion on single core serial code optimizations to produce minimal latency functions, however, our functions are inherently built to be thread safe and will scale well on multiple cores. We provide a comprehensive approach, which includes feasible algorithmic level optimizations, group level optimizations, and field element optimizations along with a discussion of potential resource use vs. speed tradeoffs as applicable.

4.1 Algorithmic Level Optimizations

As previously discussed, ECDSA sign algorithm requires the generation of (k, k^{-1}), a per message one time secret number and its inverse modulo n. To properly generate (k, k^{-1}) could take around 20,000 cycles or higher in a typical implementation. However, this part of the process does not depend on the contents of the message to be signed. In use-cases where it is important to reduce the latency of signing a message (i.e. cycles or time taken to return a signature after a request to sign a message is issued), (k, k^{-1}) can be pre-computed, per FIPS-186-4 Section 6.3, using a number of secure methods. How and where the (k, k^{-1}) are pre-computed and safely managed are implementation dependent. In systems with potential idle time, they can be calculated on the same core at a lower priority and managed as opaque objects or in implementation specific formats. For systems that process a large number of sign operations in bulk, they can be processed in their own assigned core and managed appropriately. Implementers must be cognizant of side-attack techniques and must have secure access methods for stored values of (k, k^{-1}).

For the verification algorithm, if the use case calls for verifying signatures for multiple messages under the same or different Public Keys a process called Batch Verification can be used. Especially in cases where the client program cares only whether the whole batch of signatures is valid or invalid under the same algorithm, rather than which individual signatures are valid or invalid, Batch Verification can provide a substantial performance boost [33, 34].

4.2 Group Level Optimizations

The scalar multiplication consumes the bulk of the evaluation time, and must be implemented carefully to ensure that it does not inadvertently leak information about the secret scalar. For ECDSA sign operation of prime curves, the point used in the multiplication phase is always the base point G, which is a known value and can be pre-calculated. Thus, to reduce the latency of the point multiplication, Algorithm 2 can be used, which employs a fixed-base NAF windowing method [31].

INPUT: Window width w, pos integer k, $P \in E(Fq)$
OUTPUT: $A = kP$

1. Pre-computation: Compute $P_i = 2^{wi} P$, $0 \le i \le \lceil ((t+1)/w) \rceil$
2. Compute NAF(k) = $\sum_{i=0}^{l-1} ki2^i$
3. $d \leftarrow \lceil (l/w) \rceil$
4. $(k_{l-1}, \ldots, k_1, k_0) = K_{d-1} \| \cdots \| K_1 \| K_0$ each K_i is a $\{0, \pm 1\}$-string of length w
5. If w is even then $I \leftarrow (2^{w+1}-2)/3$; else $I \leftarrow (2^{w+1}-1)/3$
6. Evaluation: $A \leftarrow \infty$, $B \leftarrow \infty$
7. For j from I down to 1 do
   - 7.1 For each i, if $K_i = j$ do: $B \leftarrow B + P_i$
   - 7.2 For each i, if $K_i = -j$ do: $B \leftarrow B - P_i$
   - 7.3 $A \leftarrow A + B$
8. Return (A)

Algorithm 2) Fixed-base NAF Windowing Method for Point Multiplication

The running time of this algorithm is approximately $(2^{w+1}/3+d-2)A$, which effectively eliminates all the doublings during the evaluation phase. Taking *P-256* curve as an example and using a window size w=4, where $d = \lceil (l/w) \rceil = 64$, the running time of the scalar multiplication is reduced to about 73 Point Addition operations (Note that d does not necessarily have to be 64, due the fact that recoding could generate an l which is not equal to m). This algorithm requires the pre-computation of 64 EC Points at a storage requirement of ~4K Bytes (64 Bytes*64). As the window size grows, the evaluation cycles decrease, but the pre-compute cycle requirements and storage size increase. There is no general rule indicating the optimal window size, and usually the best choice depends on the use-case. For *P-256*, window sizes of 4 and 5 facilitate a well-balanced implementation where storage is available for pre-computed points.

To be SPA resistant, it is desirable that either the scalar multiplication operation itself is regular (i.e. use a constant flow of point operations) or the underlying field operations are regular. Algorithm 2 improves both the performance and the regularity

compared to Algorithm 1, since it only uses Point Additions, rather than both Point Double and Point Add operations, which could be detected by SPA methods. Brickell, Gordon, McCurley and Wilson [35] discuss models to simplify the pre-computation in order to reduce the number of points to be stored. Taking $(K_{d-1}, \ldots, K_1, K_0)_2^w$ as the base 2^w representation of k, where $d = \lceil (m/w) \rceil$, then

$$kP = \Sigma^{d-1}_{(i=0)} K_i(2^{wi} P)$$

For each i from 0 to $d-1$, we then pre-calculate j number of points (where $j = (2^{w+1}-2)/3$ if w is even; and $j = (2^{w+1}-1)/3$ if w is odd) and store the pre-calculated Affine coordinate points (X, Y) in a two dimensional table, such as PTable[i][j]. The negative of the Y coordinate can also be stored in the table or computed on the fly depending on available storage.

Comparative evaluation of various Point Addition operations indicate that mixed coordinate addition such as Chudnovsky+Affine, which provide the result in Chudnovsky coordinates is the preferred addition method. The C + A -> C addition only requires 8 Field Multiplications and 3 Field Square operations.

Thus, we can re-write Algorithm 2 as shown in Algorithm 3 with the running time of (d)A, approximately 64 Point Additions for the Curve *P-256*. Our table includes the negative value of Y, therefore we only use a regular and constant-time Point Add operation in our inner loop. SafeSelect is an implementation specific function to select the appropriate table entry without leaking information about the secret scalar. Emilia Kasper [36] and Shay Gueron [37] provide well-written examples of performing the SafeSelect function and their code is available as Open Source within the OpenSSL code base [38].

INPUT: NAF(k), *d, pT* (Pointer to pre-computed data table)
OUTPUT: A = kP

1. Evaluation: A←∞
2. For i from *0* to *d-1* do
    2.1 SafeSelect (Pi), use Ki=j to choose the appropriate *P[i][j]* from PTable (handle –j)
    2.2 A←A + Pi
3. Return(A)

Algorithm 3) SCA Resistant Fast Fixed-base NAF Windowing Method for Point Multiplication

Algorithm 3 can be extended for use with multiple known points. In use cases where there is storage available for pre-calculated points for all the known points Algorithm 4 provides a very fast option. It should be noted that for ECDSA Verification there is no secret information that can be leaked, so if desired, faster, non-constant time versions of the underlying functions can be used.

INPUT: NAF(*u1*), NAF(*u2*), *d1, d2*, *pT1, pT2* (Pointers to pre-computed data tables)
OUTPUT: A = *u1P1+u2P2*

1. Evaluation: A←∞
2. *dmax* = max[*d1, d2*]; shorter NAF padded with 0s
3. For i from *0* to *dmax-1* do
    3.1 Select (P1i), use Ki=j from [NAF*(u1)*] to choose the appropriate *P1[i][j]* from *pT1*
    3.2 A←A + *P1i*
    3.3 Select (P2i), use Ki=j from [NAF*(u2)*] to choose the appropriate *P2[i][j]* from *pT2*
    3.4 A←A + *P2i*
4. Return (A)

Algorithm 4) Fast NAF Windowing Method for 2-Scalar Point Multiplication (both points known)

4.3 Field Level Optimizations

For ECC, performance depends directly on the implementation of the multiple precision arithmetic functions required to support the group level algorithms. All field operations are performed modulo an associated prime number; therefore support for signed integers is not necessary, which substantially simplifies the implementation of the field functions. At a minimum, multi-precision functions are needed for comparison, addition, subtraction, squaring, multiplication, modular reduction, and modular inversion.

In our *P-256* implementation, we use a structure of four field elements (i.e. type *tfep256*), each an unsigned integer of 64-bit (i.e. type *tUint64*) so that they will natively fit into 64-bit registers with 64-bit CPUs. Squaring and multiplying two 4-field element entities can result in an 8-field element result (i.e. type *tfelp256*), and Barrett Reduction, described below, may extend to 9-field elements. Any potential overflow beyond the 4 or 8 field elements (depending on the function) can be treated as a carry bit. Constant time functions can easily be written for compare, addition modulo associated prime, and subtraction modulo associated prime. Writing these functions in native or intrinsic assembly provides the most efficient handling of any carries and reduction. Implementers should note that most underlying field arithmetic functions are called a substantial number of times by the group operations; therefore they should be as optimal as possible for the underlying

architecture without affecting SCA resiliency [39].

With new 64-bit instructions available on recent processors, ultra fast multi-precision square and multiply operations can be implemented [40, 41]. Performance analysis indicates that the reduction functions for square and multiply significantly affect overall system performance. In order to reduce the latency of either the Sign or Verify operations, reductions must be optimized for the target platform. For *P-256*, there are three performance-oriented methods for reducing with *modulo* p_{256} (i.e. q):

1) Fast Reduction Modulo p_{256}, by Solinas [42];

2) Barrett Reduction [43];

3) Word-by-word Montgomery Multiplication and Reduction [44].

p_{256}, which is a Generalized Mersenne Prime is given below: (as 32-bit double words, and 64-bit quad words to fit into four field elements).

p_{256} = 0xffffffff 0x00000001 0x00000000
0x00000000 0x00000000 0xffffffff
0xffffffff 0xffffffff

p_{256} = 0xffffffff00000001 0x0000000000000000
0x00000000ffffffff 0xffffffffffffffff

The special form of this prime allows several simplifications to be made that substantially increase performance. Fast Reduction Modulo, Algorithm 5, while relatively straightforward to implement, involves concatenation of 32-bit values of the 8-field element input, which require additional processing for 64-bit implementations. However, it can be implemented with simple shift, mask, and add/subtract instructions. At Steps 11 and 12, if d_1 and d_2 are larger than p_{256}, they need to be subtracted from $2p_{256}$, otherwise from p_{256}. At Step 15, the implementer must ensure that if the addition at any point produces a value larger than p_{256}, that value is properly reduced.

INPUT: a, p_{256}
OUTPUT: r = a (mod p_{256})
1. < a*i* are 32–bit >
2. t ←(a07 || a06 || a05 || a04 || a03 || a02 || a01 || a0)
3. $s1$ ←(a15 || a14 || a13 || a12 || a11 || 0 || 0 || 0)
4. $s2$ ←(0 || a15 || a14 || a13 || a12 || 0 || 0 || 0)
5. $s3$ ←(a15 || a14 || 0 || 0 || 0 || a10 || a09 || a08)
6. $s4$ ←(a08 || a13 || a15 || a14 || a13 || a11 || a10 || a09)
7. $d1$ ←(a10 || a08 || 0 || 0 || 0 || a13 || a12 || a11)
8. $d2$ ←(a11 || a09 || 0 || 0 || a15 || a14 || a13 || a12)
9. $d3$ ←(a12 || 0 || a10 || a09 || a08 || a15 || a14 || a13)
10. $d4$ ←(a13 || 0 || a11 || a10 || a09 || 0 || a15 || a14)
11. $d1 \leftarrow 2p_{256} - d1$
12. $d2 \leftarrow 2p_{256} - d2$
13. $d3 \leftarrow p_{256} - d3$
14. $d4 \leftarrow p_{256} - d4$
15. $r \leftarrow t + 2\,s1 + 2\,s2 + s3 + s4 + d1 + d2 + d3 + d4$
16. < Reduce r mod p_{256} by subtraction of up to ten multiples of p_{256} >
17. Return (r)

Algorithm 5) Fast Reduction Modulo p_{256}

Barrett Reduction does not depend on the special form of p_{256}, and can actually be used to calculate $a \bmod p$ for any two positive integers a and p, with the requirement that for multi-precision numbers a is twice the size of p. Thus, Barrett Reduction can be used for reducing both with p and n. Even though the algorithm itself does not exploit the special form of any moduli, any multiplications with 0 value elements of the modulus can be optimized to reduce the running time. Furthermore, the divisions and the mod operations can be done with quad word shifts. To increase performance, $\mu = \lfloor b^{2k}/p \rfloor$ can be pre-calculated and stored as a constant if storage is available. For Algorithm 6 Step 5, Barrett indicates that the result will always be in the range of 0 to *3p-1* and 90% of the time no subtraction will be needed [45].

INPUT: p, $b \geq 3$, $k = \lfloor \log_b p+1 \rfloor$, $0 \leq a < b^{2k}$, and $\mu = \lfloor b^{2k}/p \rfloor$
OUTPUT: $r = a \bmod p$.
1. $q \leftarrow \lfloor a / b^{k-1} \rfloor \cdot \mu$
2. $q' \leftarrow \lfloor q / b^{k+1} \rfloor$
3. $r \leftarrow (a \bmod b^{k+1}) - (q'.\, p \bmod b^{k+1})$
4. If $r < 0$ then $r \leftarrow r + b^{k+1}$
5. While $r \geq p$ do: $r \leftarrow r - p$
6. Return (r)

Algorithm 6) Barrett Reduction modulo p

Montgomery Multiplication defined as $MultMM\ (a,b) = a.b.2^{-m} \bmod p$, replaces classical modular multiplication $a.b \bmod p$. This extends to the case when $a=b$, and can be used in place of the classical modular square $a.a \bmod p$, such as $SqrMM\ (a) = a.a.2^{-m} \bmod p$. Using m=256, s=64 (size of register/field element), implies k=4. p_{256} is an odd modulus and satisfies the equation $-1/p \bmod 2s = 1$. Thus using Algorithm 7, a word-by-word multiplication for p_{256}, we can reduce the required overall number of 64-bit multiplications to $2k^2$ (32), a substantial savings. Additionally, recall that one of the field elements of p_{256} is equal to 0, so any multiplications by this element can be ignored.

Furthermore, using the observation described by Gueron, that $t_1.(0xffffffffffffffff) = t_1.2^{64} - t_1$, any multiplication by the first 64-bit quad word can be replaced by a faster subtraction. However, using *MultMM* and *SqrMM* require that the projective coordinates are converted to Montgomery Domain

either by multiplying each coordinate by $2^m \bmod p$, or *MultMM* with $2^{2l} \bmod p$, which can be stored as a constant. A *MultMM* by 1 converts the coordinates back to the Residue Domain. These conversions increase cycle usage so implementers must strive to minimize them.

INPUT: $p < 2^l \;\; 0 \le a, b < p, \;\; l = s.k$
OUTPUT: $r = a.b.2^{-l} \bmod p$

1. $t = a.b$
2. for i 1 *to* k do
   2.1 $t_1 = t \bmod 2s$
   2.2 $t_2 = t_1 . p$
   2.3 $t_3 = (t + t_1)$
   2.4 $t = t_3 / 2s$
3. if $t \ge p$ then $r = t - p$
4. else $r = t$
5. Return (r)

Algorithm 7) Montgomery W-by-W Reduction

Table 1 and Chart 1 below show a comparison of the three methods discussed. For this comparison a lower number (i.e. low cycle count) is better.

	Reduction with p_{256}			Reduction with n_{256}
	Mul Solinas	*Mul Barrett*	*Mul MM*	*Mul Barrett*
cycles	438 (1X)	322 (0.74X)	298 (0.68X)	325

Table 1) Multiplication + Reduction Cycles

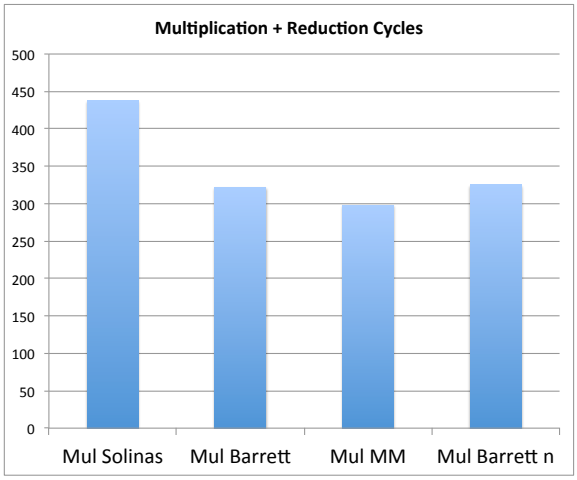

Chart 1) Multiplication + Reduction Cycles

For each operation, two random 4-field element values are multiplied, the resulting 8-field mp-number is reduced using a loop of 1000 iterations, and the median cycles are captured. Barrett Reduction can be used for both reducing with p and n, and respective median cycles for reducing with each modulus are provided in the chart. Depending on the use-case and the platform further optimizations could be performed, so implementers should performance analyze and optimize routines for their target platform.

During message signing, modular inverse operation is performed twice. The first modular inverse is required to obtain the inverse of k, k^{-1}, as discussed previously, where n is used as the modulus. The second one, where p_{256} is used as the modulus, is needed to convert from Projective coordinates to Affine coordinates at the end of the scalar multiplication operation. During verification, two modular inversions are required. First, to calculate the modular inverse of the s component of the signature (with modulus n); and, second, to convert from Projective coordinates to Affine coordinates at the end of the multi-scalar multiplication (with modulus p_{256}). While well understood, modular inverse is a costly operation and needs to be optimized to the modulus used, if possible. Implementers have several options, which include extended gcd algorithms and Fermat's Little Theorem. For verification, there is no secret information so the fastest possible algorithms without constant-time run limitations are beneficial. For sign, implementers need to ensure that no secret information is leaked with the modular inverse operation.

Results

Our solution can be run on any processor that supports x86-64 or AMD64 instructions and can be ported to other 64-bit architectures. To obtain qualitative results, performance analysis and evaluation of the proposed optimizations have been performed on a platform with Intel® Xeon® E3 1275v3 (4 core, 3.5GHz, 8M Last Level Cache) using GCC 4.9.2, with Intel Enterprise SSDs. All tests are run on a single core with both HyperThreading and Turbo turned off. Additionally, in our implementation, we use constant time functions only where they are absolutely needed to protect any secret information. Sign and Verify performance is given as operations per second (ops/sec), and higher numbers are better.

Recently, there have been substantial updates to OpenSSL *P-256* implementations [38]. NISTZ256 is a fast ECDSA *P-256* implementation included with the later versions of OpenSSL. Thus, for a reference, we compiled OpenSSL version 1.0.2 with GCC 4.9.2 on our target platform described above and obtained performance numbers using the standard *"openssl speed."* The reported number of ECDSA sign ops/sec

for curve *P-256* using OpenSSL 1.0.2 NISTZ256 implementation is 29,938 ops/sec, and verify ops/sec is 11,842. To our knowledge, this has been the highest performance, publicly available implementation of ECDSA *P-256*.

For a direct comparison we locally integrated our sign and verify functions into the *"openssl speed"* test using the same measurement and calling conventions [Note that our implementation is not submitted to OpenSSL]. With our agile algorithm both the Curve Generator Point (G) and Public Key (Q) can be treated as known points, a technique that is applicable to BGPSEC Protocol implementations. This testing mechanism shows 45,300 sign operations per sec and 31,805 verify operations per second for our implementation, *tarap256*. Signatures created with our implementation (*tarap256*) can be verified with any *P-256* compliant implementation, such as OpenSSL, and signatures created with any *P-256* compliant implementation can be verified by our implementation. Table 2 and Chart 2 summarize our results.

	ECDSA *P-256*	
	NISTZ256 measured with *openssl speed*	***tarap256*** measured with *openssl speed*
sign (ops/sec)	29,938 (1X)	45,300 (1.51X)
verify(ops/sec)	11,842 (1X)	31,805 (2.69X)

Table 2) ECDSA – NISTZ256 vs. *tarap256* Measured with OpenSSL speed

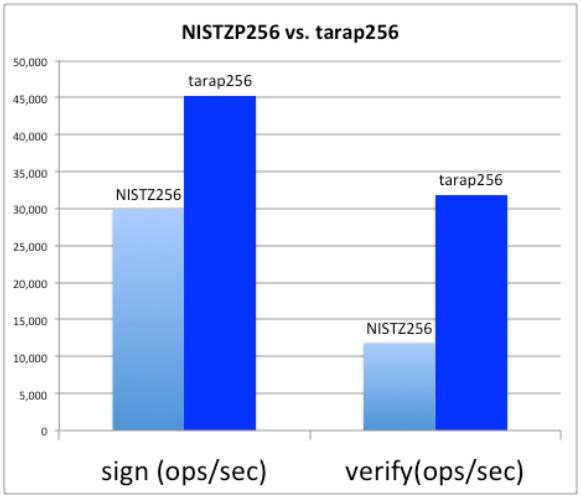

Chart 2) ECDSA – NISTZ256 vs. *tarap256* Measured with OpenSSL speed

ecdonalp256 is a well-known *P-256* implementation included in the eBACS results [46]. eBACS results include median cycles for signing and verifying 59-byte messages. Our results discussed above use the same CPU type reported on eBACS (amd64; HW+AES (306c3); 2013 Intel Xeon E3-1275 V3; 4 x 3500MHz; titan0). eBACS lists results in cycles taken. Using CPU speed as 3500MHz, we convert the latest posted eBACS cycles to operations per second which are for sign: 9,170 ops/sec and for verify: 3,830 ops/sec. We include these eBACS derived results for *ecdonaldp256* in Table 3.

There have been discussions about alternative public key algorithms and curves, such as *ed25519* and potential performance implications [47]. The signature and verify algorithms for prime curves such as *P-256* and *ed25519* are substantially different. Additionally, *P-256* and *ed25519* signatures, even tough the same size, are not compatible with each other. However, for completeness we include performance numbers for *ed25519* in Table 3 (*ed25519* on eBACS – amd64; HW+AES (306c3); 2013 Intel Xeon E3-1275 V3; 4 x 3500MHz; titan0) [42]. Since eBACS lists results in cycles taken, using CPU speed as 3500MHz, we convert the eBACS cycles for *ed25519* to operations per second which are for sign: 56,473 ops/sec and for verify: 18,920 ops/sec.

The *ed25519* algorithm does not use a per message one time random number, while *P-256* requires one to be computed. However, per FIPS-186-4 [16] the per-message random number for ECDSA can be pre-calculated without affecting the security of the operation. For a direct comparison, we implemented a version of our algorithm (*tarap256f*), which uses pre-calculated and re-coded one time random numbers and corresponding inverses. We tested this instance of our sign algorithm, *tarap256f*, using an interface similar to *ecdonalp256*. In this case, we use a message size of 64 Bytes, and a fresh message is supplied to each iteration on our Intel® Xeon® E3 3500MHz 1275v3 based platform. We compute the sign and verify ops per second using median cycles of a large number of iterations (i.e. 1,000 iterations). In this mode, with *tarap256f*, we report over 63,807 sign operations per second on a single core. The verify algorithm does not use a one time random number so this optimization is only applicable to the sign operation. These results are shown in Table 3 and Chart 3 (higher numbers better).

	ECDSA *P-256*		*c-25519*
	ecdonaldp256	*tarap256f*	*ed25519*
sign (ops/s)	9,1670 (1X)	63,807 (6.96X)	56,473 (6.16X)

Table 3) *ecdonaldp256* and *ed25519* vs. *tarap256f*

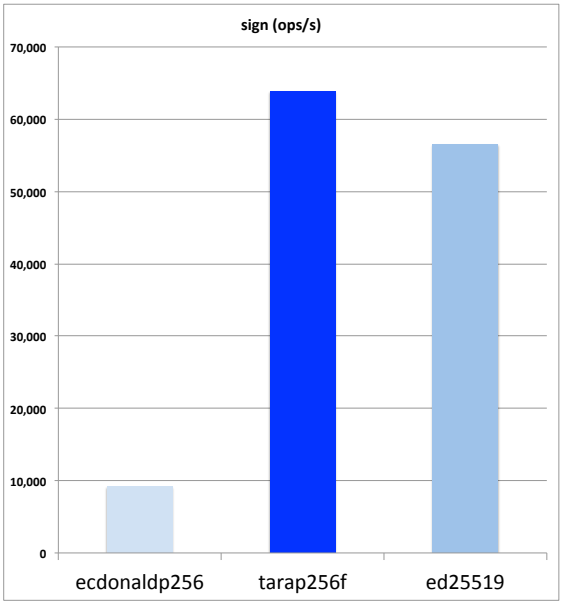

Chart 3) *ecdonaldp256* and *ed25519* vs. *tarap256f*

Conclusion

Our performance results show that it is possible to implement ultra fast and secure implementations of ECDSA for the curve *P-256*, providing 128-bits of security, on low-cost and low-power commercially available hardware. Furthermore, our work can be extended to other prime curves such as Curve *P-521*, which provides 256-bits of security. Our current implementation already includes versions of core functions that use Broadwell instructions (Intel's next generation CPU), which will show additional performance benefits when the new processors become available. Furthermore, our libraries are designed, developed, and tested to be thread-safe and can be deployed on multiple cores in parallel for additional scale and substantial performance gains.

Acknowledgements

The author would like to thank Kotikalapudi Sriram, Oliver Borchert, and Doug Montgomery for comments and suggestions during the course of this work. Also, thanks are due to anonymous reviewers for their feedback on an earlier draft of the paper.

References

1. NIST, "Framework for Improving Critical Infrastructure Cybersecurity," February 2014, http://www.nist.gov/cyberframework/upload/cybersecurity-framework-021214-final.pdf
2. K. Lougheed and Y. Rekhter, RFC 1105, "A Border Gateway Protocol (BGP)," June 1989, http://www.ietf.org/rfc/rfc1105.txt
3. Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," RFC 4271, https://www.ietf.org/rfc/rfc4271.txt
4. K. Sriram, D. Montgomery, O. Borchert, O. Kim, and R. Kuhn, "Study of BGP Peering Session Attacks and Their Impacts on Routing Performance," IEEE Journal on Selected Areas in Communications: Special issue on High-Speed Network Security, Vol. 24, No. 10, October 2006, pp. 1901-1915
5. S. Murphy, "BGP Security Vulnerabilities Analysis," RFC 4272, https://www.ietf.org/rfc/rfc4272.txt
6. Sriram, K., Montgomery, D., McPherson, D., and E. Osterweil, "Problem Definition and Classification of BGP Route Leaks", draft-ietf-grow-route-leak-problem-1 (work in progress), March 2015.
7. K. Butler, T. R. Farley, P. McDaniel, and J. Rexford, "A Survey of BGP Security Issues and Solutions," Proceedings of IEEE Vol. 98, No. 1, January 2010
8. D.R. Kuhn, K. Sriram, and D. Montgomery, "Border Gateway Protocol Security, " NIST Special Publication 800-54 (BCP document for the Telecom Industry and US Government agencies), July 2007
9. K. Sriram, O. Borchert, O. Kim, and P. Gleichmann, and D. Montgomery, "A Comparative Analysis of BGP Anomaly Detection and Robustness Algorithms," Proceedings of the Cybersecurity Applications and Technology Conference for Homeland Security (CATCH), Washington D.C., March 3-4, 2009, pp. 25-38. http://www.nist.gov/itl/antd/upload/NIST_BGP_Robustness-2.pdf
10. M. Lepinski, R. Austein, S. Bellovin, R. Bush, S. Kent, W. Kumari, D. Montgomery, K. Sriram, S. Weiler, draft-ietf-sidr-bgpsec-protocol-11, "BGPSEC Protocol

Specification," Jan 19, 2015, https://datatracker.ietf.org/doc/draft-ietf-sidr-bgpsec-protocol/

11. P. Mohapatra, J. Scudder, D.Ward, R. Bush, and R. Austein, RFC 6811, "BGP Prefix Origin Validation," January 2013
12. M. Lepinski and S. Kent, RFC 6480, "An Infrastructure to Support Secure Internet Routing," February 2012, https://tools.ietf.org/html/rfc6480
13. K. Sriram, D. Montgomery, and R. Bush, "RIB Size and CPU Workload Estimation for BGPSEC," Presentation at the IETF 91 Joint IDR/SIDR WG Meeting, November 2014. http://www.ietf.org/proceedings/91/slides/slides-91-idr-17.pdf
14. D. Johnson, A. Menezes, S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA), Certicom White Paper, 2001, http://cs.ucsb.edu/~koc/ccs130h/notes/ecdsa-cert.pdf
15. Standards for Efficient Cryptography. Elliptic Curve Cryptography, Version 1.5, draft, 2005. http://www.secg.org.
16. NIST, Digital Security Standard (DSS), FIPS PUB 186-4 July 2013, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
17. NIST, Secure Hash Standard (SHS), FIPS PUB 180-4 March 2012, http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf
18. NIST: Mathematical routines for the NIST prime elliptic curves. (April 2010), http://www.nsa.gov/ia/_files/nist-routines.pdf
19. D. McGrew, K. Igoe, M. Salter, RFC 6490, "Fundamental Elliptic Curve Cryptography Algorithms," February 2011, http://tools.ietf.org/html/rfc6090
20. Menezes, P. van Oorschot, S. Vanstone, "Handbook of Applied Cryptography," CRC Press, 1997.
21. P. Longa, C. Gebotys, "Efficient Techniques for High Speed Elliptic Curve Cryptography," Cryptographic hardware and embedded systems, CHES 2010, p80-94, 2010
22. B. Moller, "Algorithms for multi-exponentiation," Selected Areas in Cryptography, 8th Annual International Work- shop, SAC 2001 Toronto, Ontario, Canada, August 16-17, 2001, Revised Papers, volume 2259 of Lecture Notes in Computer Science, pages 165–180. Springer, 2001.
23. A. Venelli, F. Dassance, "Faster Side-Channel Resistant Elliptic Curve Scalar Multiplication," Arithmetic, Geometry, Cryptography and Coding Theory 2009, volume 521 of Contemporary Mathematics, pages 29–40. American Mathematical Society, 2010.
24. M. Rivain, "Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves" IACR Cryptology ePrint Archive 2011, p.338, 2011
25. P. C. Kocher, " Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and other Systems," Advances in Cryptology – CRYPTO '96 vol. 1109 of Lecture Notes in Computer Science, pp. 104–113, 1996
26. P. C. Kocher, J. Jaffe, B. Jun, "Differential Power Analysis," Advances in Cryptology – CRYPTO '99 vol. 1666 of Lecture Notes in Computer Science, pp. 388–397, 1999
27. E. Brier, M. Joye, "Weierstraß Elliptic Curves and Side-Channel Attacks,",Public Key Cryptography – PKC 2002, volume 2274 of Lecture Notes in Computer Science, pages 335–345. Springer, 2002.
28. J. Coron, "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems," CHES'99, LNCS 1717, pp. 292–302, Springer-Verlag, 1999.
29. P. Y. Liardet, N. P. Smart, "Preventing SPA/DPA in ECC Systems Using the Jacobi Form," In Cryptographic Hardware and Embedded Systems – CHES 2001, pp. 401–411, 2001
30. M. Joye, J. Quisquater, "Hessian Elliptic Curves and Side-channel Attacks," Cryptographic Hardware and Embedded Systems – CHES 2001 pp. 412–420, 2001
31. K. Okeya, T. Takagi, "The Width-w NAF Method Provides Small Memory and Fast Elliptic Scalar Multiplications Secure against Side Channel Attacks," Topics in Cryptology, The Cryptographers' Track at the RSA Conference – CT-RSA 2003, volume 2612 of Lecture Notes in Computer

Science, pages 328–342. Springer, 2003.

32. M. Joye, M. Tunstall, “Exponent Recoding and Regular Exponentiation Algorithms,” Progress in Cryptology, Second International Conference on Cryptology in Africa –AFRICACRYPT 2009, volume 5580 of Lecture Notes in Computer Science, pp 334–349. Springer, 2009.
33. S. Karati, A. Das, D. R. Chowdhury, B. Bellur, D. Bhattacharya, A. Iyer, “New algorithms for batch verification of standard ECDSA signatures,” J. Cryptographic Engineering 4(4): 237-258, 2014
34. S. Karati, A. Das, “Faster Batch Verification of Standard ECDSA Signatures Using Summation Polynomials,” ACNS 2014: 438-456, 2014
35. E. F Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson, “Fast Exponentiation with Precomputation,” Advances in Cryptology – Proceedings of Eurocrypt ’92, Vol 658, pp 200-207, 1992
36. E. Käsper, “ Fast Elliptic Curve Cryptography in OpenSSL,” Financial Cryptography and Data Security. LNCS, vol. 7126, pp. 27--39. Springer, Heidelberg, 2012.
37. S. Gueron, V. Krasnov, “Fast Prime Field Elliptic Curve Cryptography with 256 Bit Primes,” Journal of Cryptographic Engineering, November 2014. https://eprint.iacr.org/2013/816.pdf
38. S. Gueron, V. Krasnov, [PATCH] “Fast and side channel protected implementation of the NIST P-256 Elliptic Curve for x86-64 platforms,” OpenSSL patch, October 2013
39. P. Longa, A. Miri, “Fast and flexible elliptic curve point arithmetic over prime fields,” Computers, IEEE Transactions on 57 (3), pp289-302, 2008
40. E. Ozturk, J. Guilford, V. Gopal, W. Feghali, “New Instructions Supporting Large Integer Arithmetic on Intel® Architecture Processors,” Intel White Paper, August 2012 http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/ia-large-integer-arithmetic-paper.html
41. E. Ozturk, J. Guilford, V. Gopal, “Large Integer Squaring on Intel® Architecture Processors, ” Intel White Paper, January 2013. http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/large-integer-squaring-ia-paper.html?wapkw=large+integer+squaring+on+intel®+architecture+processors
42. 21. J. A. Solinas, “Generalized Mersenne Numbers,” Technical Report, Center for Applied Cryptographic Research. University of Waterloo, 1999.
43. P. L. Montgomery, “Modular Multiplication without Trial Division,” Mathematics of Computation, vol. 44, pp. 519—521, 1985
44. J. F. Dhem, “Modified version of the Barrett algorithm,” Technical report, 1994.
45. P. Barrett, “Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor,” In Proc. CRYPTO’86, pages 311–323, 1986.
46. D. J. Bernstein and T. Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. http://bench.cr.yp.to, accessed 15 March 2015
47. D.J. Bernstein, T. Lange, P. Schwabe, B-Y. Yang, "High-Speed High-Security Signatures", Journal of Cryptographic Engineering, Vol. 2, September 26, 2011