Faster Binary Curve Software: A Case Study[⋆]

Billy Bob Brumley

Department of Pervasive Computing
Tampere University of Technology, Finland
billy.brumley@tut.fi

Abstract. For decades, elliptic curves over binary fields appear in numerous standards including those mandated by NIST, SECG, and ANSI X9.62. Many popular security protocols such as TLS explicitly support these named curves, along with implementations of those protocols such as OpenSSL and NSS. Over the past few years, research in improving the performance and/or security of these named curve implementations has pushed forward the state-of-the-art: e.g. projective lambda coordinates (Oliveira et al.) and commodity microprocessors featuring carryless multiplication instructions for native polynomial arithmetic (Intel, ARM, Qualcomm). This work aggregates some of these new techniques as well as classical ones to bring an existing library closer to the state-of-the art. Using OpenSSL as a case study to establish the practical impact of these techniques on real systems, results show significant performance improvements while at the same time adhering to the existing software architecture.

Keywords: applied cryptography, public key cryptography, elliptic curve cryptography, OpenSSL

1 Introduction

Of the many types of public key cryptography available, elliptic curve cryptography (ECC) offers many attractive advantages – the main being the small size of private and public keys. Furthermore, since the introduction in the 1990s ECC has undergone extensive standardization – including NIST [20, D.1.3], SECG, and ANSI X9.62. Generally, these standardized curves come in two flavors:

– Elliptic curves over prime fields $\mathbb{F}_p$;
– Elliptic curves over binary fields $\mathbb{F}_{2^m}$.

For elliptic curves in software, elliptic curves over $\mathbb{F}_p$ are a more popular choice for performance reasons because the finite field arithmetic is easier to implement efficiently since most microprocessors feature native integer multiplication instructions as a building block for multi-precision arithmetic. Recognizing this

⋆ Supported in part by TEKES grant 3772/31/2014 Cyber Trust and COST Action IC1306.

practical limitation, academic efforts for high speed ECC placed more research efforts on elliptic curves over $\mathbb{F}_p$.

For elliptic curves over $\mathbb{F}_{2^m}$, historically software needed to revert to primitive table lookup methods for finite field arithmetic since it was uncommon (to say the least) to feature a native polynomial multiplication instruction. However, this trend shifted roughly five years ago when chip makers such as Intel, ARM, and Qualcomm started introducing such instructions into the Instruction Set Architecture (ISA) for commodity microprocessors. This left a gap in research for high speed ECC software for elliptic curves over $\mathbb{F}_{2^m}$ – for example, there were really no major innovations in projective coordinate systems since López and Dahab in 1999 [17]. Oliveira et al. changed that recently in 2014 with λ-projective coordinates [21].

This gap in academic research also left a gap in practical elliptic curve software libraries. The best example of this is OpenSSL, which – since introducing ECC support in 2005 – has seen no new optimizations for elliptic curves over $\mathbb{F}_{2^m}$, despite heavy optimizations for elliptic curves over $\mathbb{F}_p$.

The goal of this paper is to fill that gap and measure the real world impact of these new optimizations for elliptic curves over $\mathbb{F}_{2^m}$. The resulting OpenSSL source code patches yield performance improvements that remarkably approach 6-fold in some cases. Section 2 gives background on binary elliptic curves and discusses various coordinate systems. Section 3 gives an overview of the ECC software architecture within OpenSSL. Section 4 discusses the optimizations implemented in this paper, and gives benchmarking results. Section 5 draws conclusions.

2 Binary Elliptic Curves

For a finite field $\mathbb{F}_{2^m}$, fix curve coefficients $a_2, a_6 \in \mathbb{F}_{2^m}$ and all of the (x, y) solutions to the equation

$$E : y^2 + xy = x^3 + a_2x^2 + a_6$$

over $\mathbb{F}_{2^m}$ for $x, y \in \mathbb{F}_{2^m}$ along with the identity element (∞, point at infinity) form a finite Abelian group relevant to applied cryptography. The majority of standardized curves of this form further restrict the values that curve coefficients a_2 and a_6 can take for efficiency reasons. Generally, there are two major types.

Pseudo-random curves. These curves fix $a_2 = 1$ and $a_6 \in \mathbb{F}_{2^m}$ derived pseudorandomly. Curves of this type include, but are not limited to: B-163, B-233, B-283, B-409, and B-571.

Koblitz curves. These curves [15] fix $a_2 \in \mathbb{F}_2$ and $a_6 = 1$. Curves of this type include, but are not limited to: K-163, K-233, K-283, K-409, K-571, and sect239k1.

2.1 Scalar Multiplication

Take ℓ-bit scalar $k \in \mathbb{Z}$ where k_i denotes bit i of k and a point $P \in E$. Then the scalar multiplication result kP satisfies the following formula.

$$kP = \sum_{i=0}^{\ell-1} k_i 2^i P$$

This is the classical way to compute scalar multiplication, scanning the bits of k from MSB to LSB (or vice-versa): double-and-add, where the cost – defined w.r.t. elliptic curve operations – is $\ell - 1$ point doublings and the number of point additions is equal to the weight of k. That is, one point addition for each non-zero digit of k.

Scalar multiplication is the performance benchmark for ECC. Its speed determines the efficiency of cryptosystems like ECDSA and ECDH. Since performance is such a driving force in applied cryptography, there is no shortage of research on improving the efficiency of scalar multiplication. While the majority of these methods are beyond the scope of this paper, a few specific methods that OpenSSL employs will be discussed later (otherwise, see e.g. [11, 3.3] for a good survey).

2.2 Coordinate Systems

Since scalar multiplication breaks down into a sequence of elliptic curve point doublings and additions, the cost of these operations is critical for performance. One way to improve the efficiency of these operations is by considering different coordinate systems – the aim being to reduce the number of expensive finite field inversions. A discussion on relevant coordinate systems for elliptic curves over $\mathbb{F}_{2^m}$ follows.

Affine coordinates. The textbook method to perform addition and doubling of points on elliptic curves over $\mathbb{F}_{2^m}$ – defining the group law – is using affine coordinates [8, 13.3.1.a]. Here the inverse of $P = (x_1, y_1)$ is $-P = (x_1, x_1 + y_1)$.

Addition. Let $P = (x_1, y_1)$, $Q = (x_2, y_2)$ such that $P \neq \pm Q$. Then $P + Q = (x_3, y_3)$ is given by

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + x_1 + x_2 + a_2 \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \\ \lambda &= \frac{y_1 + y_2}{x_1 + x_2} \end{aligned}$$

Doubling. Let $P = (x_1, y_1)$ then $2P = (x_3, y_3)$, where

$$\begin{aligned} x_3 &= \lambda^2 + \lambda + a_2 \\ y_3 &= \lambda(x_1 + x_3) + x_3 + y_1 \\ \lambda &= x_1 + \frac{y_1}{x_1} \end{aligned}$$

López-Dahab coordinates. Elliptic curve operations using affine coordinates require finite field inversions (to compute λ), often an expensive operation. To eliminate these inversions, projective coordinates introduce an additional coordinate to represent points on a projective equation. While projective coordinates come in many different flavors, López and Dahab introduce a popular one for elliptic curves over $\mathbb{F}_{2^m}$ [17]. Consider the projective equation

$$Y^2 + XYZ = X^3Z + a_2X^2Z^2 + a_6Z^4.$$

The LD-projective point $(X_1 : Y_1 : Z_1)$ corresponds to the affine point $(X_1/Z_1, Y_1/Z_1^2)$ when $Z_1 \neq 0$ and the point at infinity otherwise. Here the inverse of $(X_1 : Y_1 : Z_1)$ is $(X_1 : X_1Z_1 + Y_1 : Z_1)$.

Mixed addition. A sum $P + Q$ is often more efficient to compute if one operand is in affine and the other in projective – mixed addition, colloquially. The reason this makes sense algorithmically is that for scalar multiplication, the accumulator point will undergo projective doublings and additions, but the second operand for additions remains static in affine coordinates. For LD coordinates, Al-Daoud give a slightly more efficient formula for mixed addition [1]. Let $P = (X_1 : Y_1 : Z_1)$, $Q = (x_2, y_2)$ such that $P \neq \pm Q$. Then $P + Q = (X_3 : Y_3 : Z_3)$ is given by

$$\begin{aligned}
&A = Y_1 + y_2Z_1^2, B = X_1 + x_2Z_1, C = BZ_1,\\
&Z_3 = C^2, D = x_2Z_3, X_3 = A^2 + C(A + B^2 + a_2C),\\
&Y_3 = (D + X_3)(AC + Z_3) + (y_2 + x_2)Z_3^2
\end{aligned}$$

Doubling. Let $P = (X_1 : Y_1 : Z_1)$ then $2P = (X_3 : Y_3 : Z_3)$, where

$$\begin{aligned}
&A = Z_1^2, B = a_6A^2, C = X_1^2, Z_3 = AC, X_3 = C^2 + B,\\
&Y_3 = (Y_1^2 + a_2Z_3 + B)X_3 + Z_3B
\end{aligned}$$

Lambda coordinates. The λ-affine representation [14, Sec. 2] of a short affine point $P = (x, y)$ is (x, λ) where $\lambda = x + y/x$ – note that indeed, λ is the slope from the affine doubling formula. Historically, λ-affine coordinates saw little use due to being out-performed by LD coordinates in most cases. Recently, Oliveira et al. introduce a λ-projective system [21], the performance of which remarkably eclipses LD coordinates. Consider the projective equation

$$(L^2 + LZ + a_2Z^2)X^2 = X^4 + a_6Z^4.$$

The λ-projective point $(X_1 : L_1 : Z_1)$ corresponds to the λ-affine point $(X_1/Z_1, L_1/Z_1)$ when $Z_1 \neq 0$ and the point at infinity otherwise. Here the inverse of $(X_1 : L_1 : Z_1)$ is $(X_1 : L_1 + Z_1 : Z_1)$.

Mixed addition. Let $P = (X_1 : L_1 : Z_1)$, $Q = (x_2, \lambda_2)$ such that $P \neq \pm Q$. In this case the formula for $P + Q = (X_3 : L_3 : Z_3)$ is given in [21], but from the implementation perspective the restriction $P \neq \pm Q$ is problematic – what is actually needed for implementation is an algorithmic solution, depicted in Fig. 2 (Appx. A). Indeed, this algorithm handles these corner cases to ensure correct computation for all inputs. The algorithm expects P and Q in λ-projective and λ-affine coordinates, respectively – i.e. P would be the accumulator in a scalar multiplication routine.

Doubling. Let $P = (X_1 : L_1 : Z_1)$ then Fig. 3 (Appx. A) depicts the algorithm to compute $2P = (X_3 : L_3 : Z_3)$. The input and output are both in λ-projective coordinates.

General addition. From the ECC implementation perspective, sometimes the sum of two projective points is required – e.g. in the (online or offline) precomputation step for scalar multiplication. Since that will be the case later in this paper, Fig. 4 (Appx. A) depicts the algorithm to compute $P+Q = (X_3 : L_3 : Z_3)$ where $P = (X_1 : L_1 : Z_1)$ and $Q = (X_2 : L_2 : Z_2)$ – again, compensating for the cases when $Q = P$ or $Q = -P$.

Computational costs. To conclude this section, the goal is to select the coordinate system that has the lowest computational cost w.r.t. finite field operations – inversions, multiplications, and squarings. The cost of inversions is usually very high (e.g. at least eight times that of a multiplication), so affine coordinates are not immediately useful in that respect. Table 1 summarizes the costs for the previously discussed coordinate systems, assuming $a_2 \in \mathbb{F}_2$. Based on these numbers, clearly λ-projective coordinates have an efficiency advantage.

Table 1. Computational costs of elliptic curve operations in various coordinate systems w.r.t. finite field inversions (I), multiplications (M), and squarings (S)

Coordinates	double	add	negate
affine	$1I + 2M + 1S$	$1I + 2M + 1S$	–
LD-projective (mixed)	$4M + 5S$	$8M + 5S$	$1M$
λ-projective (mixed)	$4M + 4S$	$8M + 2S$	–
λ-projective	$4M + 4S$	$11M + 2S$	–

3 ECC in OpenSSL

OpenSSL integrated support for elliptic curves in 2005. At a high level, the ECC portion of OpenSSL generically supports elliptic curves in short Weierstrass form over $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$, only the latter being immediately relevant to this paper. What follows is a discussion on the ECC portion of OpenSSL, from the architecture level and later to the concrete methods used for binary curve arithmetic.

3.1 Generic Curve Support

When an application or the library instantiates a curve, an `EC_GROUP` structure holds the curve parameters (e.g. finite field, curve coefficients, curve order, generator point, etc.) and an `EC_METHOD` structure controls computations and operations on the particular curve. The latter structure is critical to this paper – a description follows.

The `EC_METHOD` structure (overview in Fig. 1) contains a set of function pointers to carry out elliptic curve operations (e.g. double, add) as well as various interface and conversion operations (e.g. extracting points to strings). The reason this structure exists is modularity – it allows elliptic curves to be treated mostly generically from the interface perspective, but abstracts away implementation aspects of a particular curve. The simplest example of this is the library supporting both curves over $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$ – the group law for these types of curves is entirely different, but both can be supported with their own `EC_METHOD` by setting function pointers such as `add` and `dbl` to distinct functions for their corresponding curve types. Conceptually, one way to view this is analogous with object-oriented programming where the function pointers correspond to class methods.

What follows is a brief discussion of function pointers that are relevant to this work, to help understand implementation considerations in later sections. Method `point_set_affine_coordinates` sets the coordinates of the `EC_POINT` given the short affine coordinates `x` and `y`. The `get` method is the inverse, returning the short affine coordinates of the point. Methods `add` and `dbl` compute elliptic curve additions and doublings, respectively, while invert sets P to $-P$. Method `is_on_curve` checks if the point satisfies the curve equation; `make_affine` converts a single point from projective to affine coordinates, while `points_make_affine` does the same but for an arbitrary number of points. Scalar multiplication method `mul` computes

$$aG + \sum_{i=0}^{n} b_i P_i$$

hence a fully generic multi-scalar multiplication supporting an arbitrary number of scalars and corresponding points. Since G is fixed for each `EC_GROUP`, some scalar multiplication techniques precompute various multiples of points to speed up scalar multiplication; `precompute_mult` carries out such precomputation and `have_precompute_mult` checks if said precomputation is present. Finally, `field_mul`, `field_sqr`, and `field_div` compute finite field multiplications, squarings, and divisions for the particular field in `EC_GROUP` – finite field parameters often have a special form that allow e.g. fast modular reduction, so having dedicated function pointers offers the implementer a convenient way to integrate such optimizations.

```
struct ec_method_st {
...
    int (*point_set_affine_coordinates) (const EC_GROUP *, EC_POINT *,
                                         const BIGNUM *x, const BIGNUM *y,
                                         BN_CTX *);
    int (*point_get_affine_coordinates) (const EC_GROUP *, const EC_POINT *,
                                         BIGNUM *x, BIGNUM *y, BN_CTX *);
...
    int (*add) (const EC_GROUP *, EC_POINT *r, const EC_POINT *a,
                const EC_POINT *b, BN_CTX *);
    int (*dbl) (const EC_GROUP *, EC_POINT *r, const EC_POINT *a, BN_CTX *);
    int (*invert) (const EC_GROUP *, EC_POINT *, BN_CTX *);
...
    int (*is_on_curve) (const EC_GROUP *, const EC_POINT *, BN_CTX *);
...
    int (*make_affine) (const EC_GROUP *, EC_POINT *, BN_CTX *);
    int (*points_make_affine) (const EC_GROUP *, size_t num, EC_POINT *[],
                               BN_CTX *);
...
    int (*mul) (const EC_GROUP *group, EC_POINT *r, const BIGNUM *scalar,
                size_t num, const EC_POINT *points[], const BIGNUM *scalars[],
                BN_CTX *);
    int (*precompute_mult) (EC_GROUP *group, BN_CTX *);
    int (*have_precompute_mult) (const EC_GROUP *group);
    int (*field_mul) (const EC_GROUP *, BIGNUM *r, const BIGNUM *a,
                      const BIGNUM *b, BN_CTX *);
    int (*field_sqr) (const EC_GROUP *, BIGNUM *r, const BIGNUM *a, BN_CTX *);
    int (*field_div) (const EC_GROUP *, BIGNUM *r, const BIGNUM *a,
                      const BIGNUM *b, BN_CTX *);
...
} /* EC_METHOD */ ;
```

Fig. 1. OpenSSL's method structure for elliptic curves

3.2 Binary Curve Support

Since ECC integration in 2005, various `EC_METHOD` implementations started appearing in the OpenSSL code base for curves over $\mathbb{F}_p$ to optimize performance and/or security. For example, fast modular reduction routines for NIST curves P-192, P-224, P-256, P-384, and P-521; also fast and side-channel secure P-224 [13] and P-256 [10]. In contrast, for elliptic curves over $\mathbb{F}_{2^m}$ there remains only a single default `EC_METHOD` – an implementation of IEEE P1363 [12, A.10.2] that uses affine coordinates for elliptic curve point additions and doublings. To summarize, there is comparatively little to no optimization (such as projective coordinates) of binary curve operations in OpenSSL.

3.3 Finite Field Arithmetic

Software performance wise, elliptic curves over $\mathbb{F}_{2^m}$ historically lag behind those over $\mathbb{F}_p$ since most microprocessors feature integer word multiplication instructions, making the finite field multiplications more efficient in $\mathbb{F}_p$. Over the past few years, however, that trend is shifting as chips start to feature polynomial multiplication instructions – carryless multiplication, colloquially – suitable for $\mathbb{F}_{2^m}$ finite field multiplications. Some examples include:

- Intel's `pclmulqdq` instruction for 64-bit multiplication (2010);
- ARM's `vmull.p64` instruction on ARMv8 for 64-bit multiplication (2013);
- Qualcomm's `pmpyw` instruction [2, Sec. 4.1] on Hexagon DSP for 32-bit multiplication (2010).

OpenSSL integrated support for `pclmulqdq` in 2011. The implementation of polynomial multiplication is the Karatsuba method, and for chips featuring `pclmulqdq` the last level of recursion computes the product of two 128-bit polynomials – with three 64-bit multiplications, also using Karatsuba. In summary, OpenSSL has the potential for fast binary ECC since there is some acceleration of finite field operations, but currently lacks optimization of elliptic curve operations.

3.4 Scalar Multiplication

In OpenSSL, two scalar multiplication implementations are relevant for elliptic curves over $\mathbb{F}_{2^m}$.

Montgomery's Ladder. Building on their projective coordinate system result, López and Dahab combine their result with Montgomery's Ladder to yield an efficient scalar multiplication routine [17, Sec. 4.2]. OpenSSL has a fairly direct translation of their algorithm in function `ec_GF2m_montgomery_point_multiply`, taking remarkably six finite field multiplications and five squarings per scalar bit.

Interleaving. By default, if the `mul` scalar multiplication function pointer is not set OpenSSL uses Möller's interleaving with NAF splitting [18,19] in function `ec_wNAF_mul`. If precomputation exists, this implementation dramatically decreases the number of point doublings by precomputing small multiples of $2^i G$ (e.g. $i = 0, 8, 16, 24, \ldots$). If instead no precomputation is available, the NAF splitting goes away and the algorithm is then a fairly standard multi-scalar multiplication method with signed digits (NAF).

Code path. As previously discussed, the sole `EC_METHOD` for elliptic curves over $\mathbb{F}_{2^m}$ uses short affine coordinates to implement the `add` and `dbl` function pointers – the scalar multiplication function pointer `mul` is different, however. The implementation is a short wrapper that looks at the number of scalar arguments:

– If the total number of scalars involved is more than two, or a single scalar multiple of the generator *with* precomputation, the wrapper calls `ec_wNAF_mul`.
– Otherwise, the wrapper iterates `ec_GF2m_montgomery_point_multiply`.

4 Improvements

This section discusses the implemented changes to the OpenSSL code base to achieve significantly better performance for scalar multiplication with elliptic curves over $\mathbb{F}_{2^m}$. This improved efficiency is then reflected in the timings for ECDH and ECDSA cryptosystems within OpenSSL.

4.1 Lambda Method

The goal of this new `EC_METHOD` is to provide λ-projective coordinate support but at the same time utilize the existing `ec_wNAF_mul` multi-scalar multiplication function. Some of the implementation considerations are as follows.

– For offline or online precomputation, `ec_wNAF_mul` calls function pointer `add` where both operands are potentially in projective form. So the new method implements `add` as a small wrapper that calls to either an implementation of Fig. 2 or Fig. 4 – i.e. the wrapper uses the more efficient formula when it can. Function pointer `dbl` is a direct translation of Fig. 3.
– After the precomputation stage, `ec_wNAF_mul` calls `points_make_affine` function pointer so as the scalar multiplication routine executes it can use more efficient mixed coordinate point additions. But in this case, the implementation converts from λ-projective to λ-affine.
– To handle negative scalar digits, `ec_wNAF_mul` tracks the sign of the accumulator point with a flag and inverts both the accumulator and flag as necessary. So in fact λ-projective coordinates are beneficial over LD coordinates in this regard, on average saving half a finite field multiplication per digit when implementing the `invert` function pointer.

– Function pointers `point_set_affine_coordinates` (and `get`) are critical to maintain interoperability – when a cryptosystem extracts the result of scalar multiplication, it needs to be in short affine coordinates. So for `set` this implementation converts from short affine to λ-affine, and `get` converts from λ-projective to short affine to maintain compatibility.

The resulting patch to the OpenSSL code base for this method is fairly independent and non-intrusive.

4.2 Finite Field Squaring

While OpenSSL has more efficient finite field multiplications with `pclmulqdq`, squarings are still done with legacy table lookups. This modification inserts the assembly instructions to perform squaring more efficiently using `pclmulqdq`, requiring one instruction per word since squaring is an $\mathbb{F}_2$-linear operation.

4.3 Side-Channel Countermeasures

Historically, OpenSSL is a popular target for side-channel attacks that target implementation and execution aspects that leak critical secret state through e.g. latency measurements. A brief discussion on side-channel considerations follows.

Timing attacks. Previous timing attacks against OpenSSL's ECC implementation target traditional, insecure table lookups and irregular scalar encodings [7]. Although countermeasures and patches are publicly available [5], they have not been integrated into the OpenSSL codebase as of this writing.

Bug attacks. Introduced by Biham et al. [3], bug attacks target intentional backdoors in implementations of cryptographic hardware that trigger with low enough probability to go undetected by random test vectors. They give applications to public key cryptography, using a hypothetical malicious integer word multiplication instruction as an example and show how to recover a private key with cleverly chosen inputs. In [6], the first practical bug attack targets instead a real world software defect in OpenSSL and uses it to recover private keys.

Originally proposed as a Differential Power Analysis (DPA) countermeasure, Coron's randomized projective coordinates [9, Sec. 5.3] is an extremely effective and efficient countermeasure against bug attacks. At a high level, the idea is to select a random representative from the set of projective points that map to the same affine point. While outlined for canonical projective coordinates, the exact steps to select this representative depend on the relationship between the projective and affine point.

For λ-projective coordinates, the λ-projective point $(X : L : Z)$ is in fact equivalent to $(\beta X : \beta L : \beta Z)$ for all $\beta \in \mathbb{F}_{2^m} \setminus \{0\}$. This is easy to see since the λ-affine point corresponding to $(X : L : Z)$ is $(X/Z, L/Z)$, so $(\beta X : \beta L : \beta Z) \mapsto$

$((\beta X)/(\beta Z), (\beta L)/(\beta Z)) = (X/Z, L/Z)$ for all $\beta \in \mathbb{F}_{2^m} \setminus \{0\}$ – i.e. yielding the same λ-affine point.

This randomization essentially makes the state of the scalar multiplication algorithm unpredictable, hence the iterative approach needed for bug attacks is no longer feasible. For this work, we implement this with a random β chosen at the start of the `ec_wNAF_mul` main loop, when the accumulator is initialized – i.e. randomize once per scalar multiplication.

4.4 Timings

The benchmarking environment in this section is an Intel Celeron 2955U 1.40GHz (ft. `pclmulqdq`) running 64-bit Ubuntu 14.04 with 2GB of memory. Timings are with OpenSSL's own benchmarking utility, `openssl speed` with options `ecdh` and `ecdsa`. The OpenSSL version is `1.1.0-dev`, git branch `OpenSSL-master` tip[1].

ECDH results. For ECDH, Tbl. 2 shows there is no significant change after the modifications. This is rather predictable since the code path for ECDH on stock OpenSSL executes the LD version of Montgomery's Ladder that is already fairly efficient. On the bright side, deprecating the stock code and introducing λ-projective coordinates does not hurt the average performance over all curves.

Table 2. ECDH operations per second

curve	stock	modified	gain
nistk163	2107.7	2022.6	-4.0%
nistk233	1675.2	1670.2	-0.3%
nistk283	929.3	921.0	-0.9%
nistk409	589.5	563.8	-4.4%
nistk571	248.7	244.9	-1.5%
nistb163	2043.9	2011.4	-1.6%
nistb233	1600.9	1640.6	2.5%
nistb283	891.6	903.9	1.4%
nistb409	551.9	559.4	1.4%
nistb571	229.1	243.5	6.3%

ECDSA results. The ECDSA numbers tell quite a different story, shown in Tbl. 3. The gains are fairly staggering – from roughly a 3 to 6 fold performance improvement for ECDSA signature generation, and roughly 1.6 to 1.8 for ECDSA signature verification.

[1] Commit `5fced2395ddfb603a50fd1bd87411e603a59dc6f` as of this writing.

Table 3. ECDSA operations per second

curve	stock (sign)	modified (sign)	gain (sign)	stock (verify)	modified (verify)	gain (verify)
nistk163	2304.1	6723.4	191.8%	1022.9	1617.6	58.1%
nistk233	1146.2	5147.5	349.1%	791.8	1313.5	65.9%
nistk283	770.6	3136.7	307.0%	442.6	744.2	68.1%
nistk409	341.0	1969.2	477.5%	280.2	456.4	62.9%
nistk571	158.2	896.0	466.4%	120.2	199.0	65.6%
nistb163	2300.3	6684.2	190.6%	983.1	1635.9	66.4%
nistb233	1174.2	5227.7	345.2%	765.0	1280.2	67.3%
nistb283	771.3	3142.4	307.4%	420.1	735.1	75.0%
nistb409	339.8	1952.7	474.7%	262.4	446.5	70.2%
nistb571	157.6	858.8	444.9%	111.1	197.7	77.9%

5 Conclusion

Leaning on recent academic results on more efficient elliptic curve operations for elliptic curves over $\mathbb{F}_{2^m}$, this work takes OpenSSL as a case study to bring the ECC portion of the library closer to state-of-the-art. This allows to measure the real world impact of these research results. For ECDH, the performance remains roughly the same but for ECDSA the performance approaches roughly an astounding 6-fold improvement. See Tbl. 4 in the appendix to get an idea of the comparative ECC performance for standardized curves over prime fields. Lastly, it is worth noting that these results can be used in tandem with curve-specific binary field arithmetic patches to compound the performance numbers – see e.g. [4].

The source code patches – available in OpenSSL's issue tracker (RT 4103) and on the `openssl-dev` mailing list[2] – are fairly non-intrusive, adhering to OpenSSL's existing software architecture and leveraging much of the code long present in the library, in particular the multi-scalar multiplication function. In conclusion, this work validates recent advances in efficient binary curve arithmetic and brings these research results to practice where they can have direct impact.

References

1. Al-Daoud, E., Mahmod, R., Rushdan, M., Kiliçman, A.: A new addition formula for elliptic curves over GF(2^n). IEEE Trans. Computers 51(8), 972–975 (2002), `http://doi.ieeecomputersociety.org/10.1109/TC.2002.1024743`
2. Avanzi, R., Brumley, B.B.: Faster 128-EEA3 and 128-EIA3 software. Cryptology ePrint Archive, Report 2013/428 (2013), `https://eprint.iacr.org/2013/428`
3. Biham, E., Carmeli, Y., Shamir, A.: Bug attacks. In: Wagner, D. (ed.) Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology

[2] `http://marc.info/?l=openssl-dev&m=144008703808363`

Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5157, pp. 221–240. Springer (2008), http://dx.doi.org/10.1007/978-3-540-85174-5_13

4. Bluhm, M., Gueron, S.: Fast software implementation of binary elliptic curve cryptography. J. Cryptographic Engineering 5(3), 215–226 (2015), http://dx.doi.org/10.1007/s13389-015-0094-1
5. Brumley, B.B.: Faster software for fast endomorphisms. In: Mangard, S., Poschmann, A.Y. (eds.) Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers. Lecture Notes in Computer Science, vol. 9064, pp. 127–140. Springer (2015), http://dx.doi.org/10.1007/978-3-319-21476-4_9
6. Brumley, B.B., Barbosa, M., Page, D., Vercauteren, F.: Practical realisation and elimination of an ECC-related software bug attack. In: Dunkelman, O. (ed.) Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7178, pp. 171–186. Springer (2012), http://dx.doi.org/10.1007/978-3-642-27954-6_11
7. Brumley, B.B., Hakala, R.M.: Cache-timing template attacks. In: Matsui, M. (ed.) Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5912, pp. 667–684. Springer (2009), http://dx.doi.org/10.1007/978-3-642-10366-7_39
8. Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., Vercauteren, F. (eds.): Handbook of elliptic and hyperelliptic curve cryptography. Discrete Mathematics and its Applications (Boca Raton), Chapman & Hall/CRC, Boca Raton, FL (2006)
9. Coron, J.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Koç and Paar [16], pp. 292–302, http://dx.doi.org/10.1007/3-540-48059-5_25
10. Gueron, S., Krasnov, V.: Fast prime field elliptic-curve cryptography with 256-bit primes. J. Cryptographic Engineering 5(2), 141–151 (2015), http://dx.doi.org/10.1007/s13389-014-0090-x
11. Hankerson, D., Menezes, A., Vanstone, S.: Guide to elliptic curve cryptography. Springer Professional Computing, Springer-Verlag, New York (2004)
12. IEEE: Standard specifications for public key cryptography. P1363 (1999)
13. Käsper, E.: Fast elliptic curve cryptography in OpenSSL. In: Danezis, G., Dietrich, S., Sako, K. (eds.) Financial Cryptography and Data Security - FC 2011 Workshops, RLCPS and WECSR 2011, Rodney Bay, St. Lucia, February 28 - March 4, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7126, pp. 27–39. Springer (2011), http://dx.doi.org/10.1007/978-3-642-29889-9_4
14. Knudsen, E.W.: Elliptic scalar multiplication using point halving. In: Lam, K., Okamoto, E., Xing, C. (eds.) Advances in Cryptology - ASIACRYPT '99, International Conference on the Theory and Applications of Cryptology and Information Security, Singapore, November 14-18, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1716, pp. 135–149. Springer (1999), http://dx.doi.org/10.1007/978-3-540-48000-6_12
15. Koblitz, N.: CM-curves with good cryptographic properties. In: Feigenbaum, J. (ed.) Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceed-

ings. Lecture Notes in Computer Science, vol. 576, pp. 279–287. Springer (1991), http://dx.doi.org/10.1007/3-540-46766-1_22

16. Koç, Ç.K., Paar, C. (eds.): Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings, Lecture Notes in Computer Science, vol. 1717. Springer (1999)
17. López, J., Dahab, R.: Fast multiplication on elliptic curves over GF(2^m) without precomputation. In: Koç and Paar [16], pp. 316–327, http://dx.doi.org/10.1007/3-540-48059-5_27
18. Möller, B.: Algorithms for multi-exponentiation. In: Vaudenay, S., Youssef, A.M. (eds.) Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16-17, 2001, Revised Papers. Lecture Notes in Computer Science, vol. 2259, pp. 165–180. Springer (2001), http://dx.doi.org/10.1007/3-540-45537-X_13
19. Möller, B.: Improved techniques for fast exponentiation. In: Lee, P.J., Lim, C.H. (eds.) Information Security and Cryptology - ICISC 2002, 5th International Conference Seoul, Korea, November 28-29, 2002, Revised Papers. Lecture Notes in Computer Science, vol. 2587, pp. 298–312. Springer (2002), http://dx.doi.org/10.1007/3-540-36552-4_21
20. NIST: Digital signature standard (DSS). FIPS 186-4, National Institute of Standards and Technology (2013), http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
21. Oliveira, T., López, J., Aranha, D.F., Rodríguez-Henríquez, F.: Two is the fastest prime: lambda coordinates for binary elliptic curves. J. Cryptographic Engineering 4(1), 3–17 (2014), http://dx.doi.org/10.1007/s13389-013-0069-z

A Algorithms

Figures 2, 3, and 4 list the relevant formulae for λ-projective coordinate arithmetic in pseudo-code form, suitable for implementation. The algorithms assume $a_2 \in \{0, 1\}$.

B Elliptic Curves Over Prime Fields

For reference, Tbl. 4 lists OpenSSL's stock performance for ECC operations for standardized curves over prime fields. The architecture and OpenSSL version are the same as for the previous performance numbers.

Table 4. Operations per second for standardized curves over prime fields

curve	ECDH	ECDSA sign	ECDSA verify
secp160r1	2309.6	7145.5	1933.9
nistp192	1987.5	6133.2	1575.3
nistp224	1414.7	4685.9	1173.2
nistp256	5817.6	9513.6	4013.5
nistp384	603.0	2141.6	504.0
nistp521	299.5	1035.6	239.8

```
**Input:** $\lambda$-projective $P = (X_1 : L_1 : Z_1)$, $\lambda$-affine $Q = (x_2, \lambda_2)$
**Output:** $\lambda$-projective $P + Q = (X_3 : L_3 : Z_3)$
if $Q = \infty$ then return $P$
if $P = \infty$ then return $(x_2 : \lambda_2 : 1)$
$t_1 \leftarrow \lambda_2 \cdot Z_1$
$t_2 \leftarrow t_1 + L_1$
$t_1 \leftarrow x_2 \cdot Z_1$
$t_0 \leftarrow t_1 + X_1$
if $t_0 = 0$ then
    if $t_2 = 0$ then return $2P$
    else return $\infty$
$t_1 \leftarrow t_1 \cdot t_2$
$t_3 \leftarrow X_1 \cdot t_2$
$t_0 \leftarrow t_0^2$
$X_3 \leftarrow t_1 \cdot t_3$
$t_3 \leftarrow Z_1 + L_1$
$t_2 \leftarrow t_0 \cdot t_2$
$t_0 \leftarrow t_0 + t_1$
$t_3 \leftarrow t_3 \cdot t_2$
$t_0 \leftarrow t_0^2$
$Z_3 \leftarrow Z_1 \cdot t_2$
$L_3 \leftarrow t_3 + t_0$
return $(X_3 : L_3 : Z_3)$
```

Fig. 2. Mixed addition of λ-projective and λ-affine points

```
**Input:** $\lambda$-projective $P = (X_1 : L_1 : Z_1)$
**Output:** $\lambda$-projective $2P = (X_3 : L_3 : Z_3)$
if $P = \infty$ then return $\infty$
$t_0 \leftarrow L_1^2$
$t_3 \leftarrow Z_1 \cdot L_1$
$t_1 \leftarrow Z_1^2$
$t_2 \leftarrow X_1 \cdot Z_1$
$t_0 \leftarrow t_3 + t_0$
if $a_2 = 1$ then $t_0 \leftarrow t_0 + t_1$
$X_3 \leftarrow t_0^2$
$Z_3 \leftarrow t_1 \cdot t_0$
$t_0 \leftarrow t_3 \cdot t_0$
$t_2 \leftarrow t_2^2$
$t_0 \leftarrow Z_3 + t_0$
$t_0 \leftarrow t_0 + X_3$
$L_3 \leftarrow t_0 + t_2$
return $(X_3 : L_3 : Z_3)$
```

Fig. 3. doubling a point in λ-projective coordinates

```
**Input:** $\lambda$-projective $P = (X_1 : L_1 : Z_1)$, $\lambda$-projective $Q = (X_2 : L_2 : Z_2)$
**Output:** $\lambda$-projective $P + Q = (X_3 : L_3 : Z_3)$
**if** $Q = \infty$ **then return** $P$
**if** $P = \infty$ **then return** $Q$
$t_3 \leftarrow X_1 \cdot Z_2$
$t_2 \leftarrow X_2 \cdot Z_1$
$t_0 \leftarrow L_2 \cdot Z_1$
$t_1 \leftarrow Z_2 \cdot L_1$
$t_0 \leftarrow t_0 + t_1$
$t_1 \leftarrow t_2 + t_3$
**if** $t_1 = 0$ **then**
    **if** $t_0 = 0$ **then return** $2P$
    **else return** $\infty$
$t_4 \leftarrow Z_1 + L_1$
$t_2 \leftarrow t_2 \cdot t_0$
$t_5 \leftarrow t_1^2$
$t_3 \leftarrow t_3 \cdot t_0$
$t_1 \leftarrow t_5 \cdot t_0$
$X_3 \leftarrow t_2 \cdot t_3$
$t_0 \leftarrow Z_2 \cdot t_1$
$t_1 \leftarrow t_5 + t_2$
$t_4 \leftarrow t_4 \cdot t_0$
$t_1 \leftarrow t_1^2$
$Z_3 \leftarrow Z_1 \cdot t_0$
$L_3 \leftarrow t_4 + t_1$
**return** $(X_3 : L_3 : Z_3)$
```

Fig. 4. Adding λ-projective points