Comparative Study of ECC Libraries for Embedded Devices

Tjerand Silde

Norwegian University of Science and Technology, Trondheim, Norway
tjerand.silde@ntnu.no

Abstract. This paper is a comparative study of several ECC libraries optimized for embedded devices. These devices have limited computing power, and often even more limited memory, which combined force us to write specialized software to achieve practical performance capabilities. However, these optimizations often lead to algorithms that are vulnerable to side-channel attacks, such as timing attacks, simple power analysis (SPA) and differential power analysis (DPA), in addition to fault attacks. In this study, we conduct a literature review and source code inspection to compare ECC libraries' implementation of security measures. We compare these libraries to a previous study done by Nascimento et. al. The libraries of interest in this paper are Bear SSL, FLECC, Micro-ECC and mbed TLS. While several papers consider Micro-ECC to be the fastest ECC library, these papers do not document the library's security. We conclude that Micro-ECC is not just fast but also secure. All libraries of interest are secure against timing attacks and SPA, and most are secure against DPA and fault attacks as well. We suggest Micro-ECC can become even more secure by using all its built in security features by default. Lastly, we present an updated table with the security of the different ECC libraries.

Keywords: ECC Libraries · Embedded Devices · Side-Channel Attacks · Fault Attacks · Countermeasures · Security Recommendations.

1 Introduction

Background. Embedded devices have limited memory and computing power. Hence, it is important that the cryptographic software used is implemented in an efficient way that has been adjusted for these platforms. There are many lightweight libraries for symmetric cryptography and hash functions designed for embedded devices; however, in this paper we will focus on asymmetric cryptography. One of the main challenges here is that asymmetric cryptography requires large integers to be secure. This makes it much harder to implement the protocols in an efficient way.

Primitives. We use asymmetric cryptography to agree upon keys used in symmetric cryptography and to sign messages – RSA and Elliptic Curve Cryptography (ECC) are the two standardized choices. The National Institute of Standards

and Technology (NIST) specifies the set of cryptographic algorithms and key sizes considered "secure" [6]: RSA is considered secure with 2048 bit keys and Elliptic Curves with 224 bit keys. The subject of this paper is ECC, which is often preferred over RSA because of its smaller key size. In particular, we will look at the scalar multiplication algorithms used in the Elliptic Curve Diffie-Hellman (ECDH) protocol, as described by NIST [5], to agree on a shared secret key, and in the Elliptic Curve Digital Signing Algorithm (ECDSA), also described by NIST [4], used to sign messages.

Side-channel attacks. Cryptographic algorithms are traditionally designed and analyzed using the black-box model, where the adversary knows the specification of the algorithm and can observe pairs of inputs and outputs from the algorithm. This is not sufficient to protect a system when it is implemented on embedded devices. In this case, the device is often controlled by the adversary, who can apply a wide range of attacks through different side-channels to break the cryptography. Paul Kocher made a major breakthrough in the 90s when he introduced side-channel attacks (SCA) against both RSA [17] and DES [18], and was able to recover the secret keys only by looking at the timing and power differences that occurred when a server was encrypting messages. Research in this area has rapidly advanced since then. This paper will study these kinds of attacks, and their countermeasures, before evaluating the security of the aforementioned ECC libraries.

Contributions. The main contributions of this paper are threefold:

1. We give the first formal side-channel security analysis of Micro-ECC [20] via code inspection, and compare the implemented security features with countermeasures documented in the literature.
2. We give a high-level side-channel security analysis of the ECC components of the full SSL/TLS libraries Bear SSL [28] and mbed TLS [3] by reviewing documentation and inspecting code.
3. We include a high-level side-channel security analysis of FLECC [37], based on the work of Wenger et al. [36], and add the analysis of all the mentioned libraries to an updated table, extending the work of Nascimento et. al. [25].

Organization. This paper is organized as follows: We first take a look at elliptic curves and how we represent and compute additions and doubling of points on elliptic curves. Next, we look at different scalar multiplication algorithms used in ECDH and ECDSA. We then survey different side-channel attacks and countermeasures, before analyzing the ECC libraries of interest. Finally, we present the results in table 1.

2 Elliptic Curve Cryptography

In this section we introduce elliptic curves, different representations of points on elliptic curves and how we can add them together. This is essential for the scalar multiplication algorithms in the next section.

2.1 Elliptic Curve Equation

A general elliptic curve E over a finite field F_p for a prime number p is usually represented in the long Weierstrass equation for variables x and y representing values in F_p:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$
$$a_1, a_2, a_3, a_4, a_5, a_6 \in F_p.$$

However, in cryptography we often use the short Weierstrass equation:

$$E : y^2 = x^3 + ax + b, a, b \in F_p.$$

The short Weierstrass equation is equivalent to the long Weierstrass equation if the prime number p is greater than 3. The discriminant $\Delta = -16 \cdot (4a^3 + 27b^2)$ is an important characteristic of an elliptic curve, and we need $\Delta \not\equiv 0 \mod p$ to be able to perform algebraic operations on the curve.

We know from the literature that all points (x, y) which satisfy the equation, together with an identity point $\mathcal{O}$, form an Abelian group. In practice this means that for all points P and Q on the elliptic curve, there is then a way to calculate the sum $P + Q$ such that the result is also on the curve.

Let $R = [k]P = P + \cdots + P$ (adding the point P to itself a number of k times). Then it is considered difficult to find the scalar k, called the (elliptic curve) discrete logarithm, given the points P and R. This is the fundamental security of ECDH and ECDSA. We describe how we add points on an elliptic curve in the next subsection.

2.2 Point Doubling and Addition Formulas

The point addition on elliptic curves is often called the chord-tangent group law. If we're going to add two points on an elliptic curve, it is possible to prove that if you draw a straight line through the two points, then the line will always intersect the elliptic curve in a third point. In the case that you are adding a point to itself, a doubling, the tangent line through the point will intersect the elliptic curve in a second point. This is a famous result from algebraic geometry.

From the short Weierstrass equation we see that the elliptic curve is symmetric around the x-axis. We define the sum of two points to be the mirror point around the x-axis of the new point you got by intersecting the line or tangent

with the elliptic curve. Algebraically, for points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, we get the sum get $P + Q = (x_3, y_3)$ as described by the following formulas:

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} \text{ if } P \neq \pm Q, \\ \\ \frac{3x_1^2 + a}{2y_1} \text{ if } P = Q, \end{cases}$$

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2, \\ y_3 &= \lambda(x_1 - x_3) - y_1. \end{aligned}$$

Note that if $P = (x_1, y_1)$ and $Q = -P = (x_1, -y_1)$, then $P + Q = \mathscr{O}$, by definition. The scalar λ is the slope of the line through P and Q. For $P \neq Q$ we just calculate the slope in the normal way, but for $P = Q$ we use implicit derivation to calculate the slope. Points that are represented by two coordinates x and y are called affine points.

To increase the speed of adding points on elliptic curves, we need to avoid the inversions that we have to do for each addition in affine coordinates. These inversions are calculated for each bit of the scalar k, when we compute $[k]P$ for some point P.

3 Projective Coordinates

We can go from affine coordinates (x, y) to projective coordinates (X, Y, Z) by setting

$$(X, Y, Z) = (x \cdot Z^m, y \cdot Z^n, Z),$$

for some Z, where m and n are positive integers. If $m = 2$ and $n = 3$, we call it Jacobian coordinates. For example, we can go from affine coordinates to projective coordinates by setting $X = x, Y = y, Z = 1$, and go from projective coordinates to affine coordinates by setting

$$x = X/Z^m, y = Y/Z^n.$$

One of the special properties about projective coordinates is that for any projective point $P = (X, Y, Z)$ and nonzero scalar λ we get a new projective point as before:

$$\lambda \cdot (X, Y, Z) \stackrel{def}{=} (\lambda^m \cdot X, \lambda^n \cdot Y, \lambda \cdot Z),$$

which represents exactly the same affine point:

$$\begin{aligned} x &= (\lambda^m \cdot X)/(\lambda \cdot Z)^m, \\ y &= (\lambda^n \cdot Y)/(\lambda \cdot Z)^n. \end{aligned}$$

When we change coordinates from affine to projective, we also have to change the way we calculate the point additions and doublings, and this way we get rid of the inversions in the addition formulas for affine coordinates. There are many different ways to do this – we'll take a look at two of the most popular and efficient algorithms. For other addition formulas, we refer to Bernstein and Lange [7].

3.1 Jacobian Formulas

When we go from affine coordinates to Jacobian coordinates, the additions and doublings can be calculated as in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 Jacobian Doubling

Input: Point $P = (X_1, Y_1, Z_1)$.

Output: Point $R = P+P = (X_3, Y_3, Z_3)$.

1: $A \leftarrow X_1 \cdot Y_1^2$
2: $B \leftarrow 2^{-1} \cdot 3 \cdot (X_1^2 - Z_1^4)$
3: $X_3 \leftarrow B^2 - 2 \cdot A$
4: $Y_3 \leftarrow B \cdot (A - X_3) - Y_1^4$
5: $Z_3 \leftarrow Y_1 \cdot Z_1$
6: **return** (X_3, Y_3, Z_3)

Algorithm 2 Jacobian Addition

Input: Points $P = (X_1, Y_1, Z_1)$ and $Q = (X_2, Y_2, Z_2)$.

Output: Point $R = P+Q = (X_3, Y_3, Z_3)$.

1: $A \leftarrow X_1 \cdot Z_2^2$
2: $B \leftarrow X_2 \cdot Z_1^2$
3: $C \leftarrow Y_1 \cdot Z_2^3$
4: $D \leftarrow Y_2 \cdot Z_1^3$
5: $E \leftarrow A - B$
6: $F \leftarrow C - D$
7: $X_3 \leftarrow F^2 - E^3 - 2 \cdot B \cdot E^2$
8: $Y_3 = F \cdot (B \cdot E^2 - X_3) - D \cdot E^3$
9: $Z_3 = Z_1 \cdot Z_2 \cdot E$
10: **return** (X_3, Y_3, Z_3)

3.2 Co-Z Formulas

When we use Jacobian coordinates, we are performing calculations on all three coordinates X, Y and Z, instead of just X and Y as is the case when using affine coordinates. This means that in order to get rid of the inversions, we have to store more intermediate data, and do several extra multiplications. This has motivated researchers to find addition-formulas that only use two out of the three coordinates in the calculations, and then do an extra calculation in the end to recover the last coordinate before going back to affine coordinates. Meloni [22] was able to come up with formulas that efficiently add two points which have the same Z-coordinates, and also output either one of the input points or the conjugate addition that have the same Z-coordinate as the sum. In other words, if the input are the points P and Q, which have the same Z-coordinate, then

the addition can produce $P + Q$ and either P' (P with updated Z-coordinate) or $P - Q$, where all these resulting points have the same Z-coordinate. This is computed as in Algorithm 3 and 4. The calculations are described in detail in Rivain [29].

Algorithm 3 Co-Z addition with update: XYCZ-ADD

Input: Points $P = (X_1, Y_1)$ and $Q = (X_2, Y_2)$.
Output: Points $P + Q$ and P' with the same Z-coordinate.

1: $A \leftarrow (X_2 - X_1)^2$
2: $B \leftarrow X_1 \cdot A$
3: $C \leftarrow X_2 \cdot A$
4: $D \leftarrow (Y_2 - Y_1)^2$
5: $E \leftarrow Y_1 \cdot (C - B)$
6: $X_3 \leftarrow D - (B + C)$
7: $Y_3 \leftarrow (Y_2 - Y_1) \cdot (B - X_3) - E$
8: $X_1' \leftarrow B$
9: $Y_1' \leftarrow E$
10: **return** $(X_3, Y_3), (X_1', Y_1')$

Algorithm 4 Co-Z conjugate addition: XYCZ-ADDC

Input: Points $P = (X_1, Y_1)$ and $Q = (X_2, Y_2)$.
Output: Points $P + Q$ and $P - Q$ with the same Z-coordinate.

1: $A \leftarrow (X_2 - X_1)^2$
2: $B \leftarrow X_1 \cdot A$
3: $C \leftarrow X_2 \cdot A$
4: $D \leftarrow (Y_2 - Y_1)^2$
5: $E \leftarrow Y_1 \cdot (C - B)$, $F \leftarrow (Y_1 + Y_2)^2$
6: $X_3 \leftarrow D - (B + C)$
7: $Y_3 \leftarrow (Y_2 - Y_1) \cdot (B - X_3) - E$
8: $X_3' \leftarrow F - (B + C)$
9: $Y_3' \leftarrow (Y_1 + Y_2) \cdot (X_3' - B) - E$
10: **return** $(X_3, Y_3), (X_3', Y_3')$

We see that the co-Z formulas only store the X and Y values of the coordinates to save memory, and no inversions are calculated. However, there are a few things we need to take care of. The first is the input to the algorithms. We need to make sure that P and Q have the same Z-coordinate, that none of them are the point of identity $\mathcal{O}$, and that $P \neq \pm Q$, that is, P and Q are neither identical nor inverses. Our goal is to compute $[k]P$ by repeatable additions. We make sure to avoid these special cases by checking that $P \neq \mathcal{O}$ and calculating $P = P'$ and $Q = 2P'$ as in Algorithm 5 when we start, to make sure that P' and Q have the same Z-coordinate.

To go from co-Z coordinates and back to affine coordinates after the scalar multiplication, we need to know all coordinates (X, Y, Z) of the projective point, hence, we need to know the value of Z. This can be calculated in Algorithm 6, with the original point P as input, in addition to a projective point P' which represents the same affine point as P, and a point Q with the same Z-coordinate as P'. Note that Algorithm 6 calculates the inverse of the Z-coordinate, and multiplies it with the inverse of the difference in the X-coordinates of P' and Q. This then calculates the inverse of the Z-coordinate of $P' + Q$. The reason for this is explained in detail in when introducing the Co-Z Montgomery Ladder in Algorithm 9.

Algorithm 5 InitialDouble

Input: Affine point $P = (x, y)$.
Output: Co-Z projective points $2P' = (X_2', Y_2')$ and $P' = (X_1', Y_1')$.

1: $T \leftarrow 3 \cdot x^2 + a$
2: $X_1' \leftarrow 4 \cdot x \cdot y^2$
3: $X_2' \leftarrow T^2 - 2 \cdot X_1'$
4: $Y_1' \leftarrow 8 \cdot y^4$
5: $Y_2' \leftarrow T \cdot (X_1' - X_2') - Y_1'$
6: **return** $(X_2', Y_2'), (X_1', Y_1')$

Algorithm 6 FinalInvZ

Input: Affine point $P = (x, y)$, and co-Z projective points $R_0 = (X_0, Y_0)$ and $Q = (X_1, Y_1)$.
Output: $Z^{-1} = 1/(Z * (X1 - X0))$.

1: $A \leftarrow X_1 - X_0$
2: $B \leftarrow A \cdot Y \cdot x$
3: $C \leftarrow B^{-1}$
4: $Z^{-1} \leftarrow C \cdot y \cdot X$
5: **return** Z^{-1}

4 Scalar Multiplication Algorithms

Using the explicit formulas for adding points on elliptic curves, we can compute the scalar multiplication $[k]P$ for a large scalar k and a elliptic curve point P. The simplest way is to just add P to itself k times; however, this can be very inefficient for large scalars k. The following algorithms all calculate $[k]P$ were the number of additions are polynomial in $\log p$. We always assume that the most significant bit of k is equal to one.

5 Double-And-Add Algorithm

The Double-And-Add algorithm is the simplest binary algorithm. Here, you can go through bit by bit in the key k, starting with the most significant bit. The most significant bit is always one, so we start by initializing the point P. For each of the other bits, we always double the intermediate sum, and if the key bit is equal to one, we also add P to our intermediate sum. At the end of the algorithm, we have calculated the product $[k]P$.

Algorithm 7 Double-And-Add Algorithm

Input: Point P, scalar $k = (k_{n-1}, k_{n-2}, \ldots, k_1, k_0)$.
Output: Point $Q = [k]P$.

1: $R \leftarrow P$
2: **for** $i = n - 2$ **downto** 0 **do**
3: $\quad R \leftarrow R + R$ # Double
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad R \leftarrow R + P$ # Add
6: $\quad$ **end if**
7: **end for**
8: **return** R

6 Montgomery Ladder

The Montgomery Ladder was developed by Montgomery [23]. The ladder is a regularized algorithm, where you always do the doubling and addition for each bit of the key, and all the operations depends on all the previous ones. No dummy operations are computed. We also note the difference $R_1 - R_0 = P$ in each round of the loop. This is the preferred scalar multiplication in practice, and the use of projective coordinates may both speed up the calculations and offer protection against various side-channel attacks.

Algorithm 8 Montgomery Ladder

Input: Point P, scalar $k = (k_{n-1}, k_{n-2}, \ldots, k_1, k_0)$.
Output: Point $Q = [k]P$.

1: $R_0 \leftarrow \mathcal{O}, R_1 \leftarrow P$
2: **for** $i = n - 1$ **downto** 0 **do**
3: $\quad R_{1-k_i} \leftarrow R_{1-k_i} + R_{k_i}$ # Add
4: $\quad R_{k_i} \leftarrow R_{k_i} + R_{k_i}$ # Double
5: **end for**
6: **return** R_0

7 Co-Z Montgomery Ladder

Goundar, Joye and Miyaji [15] noticed that the co-Z addition formulas developed by Meloni [22] could be used to perform the scalar multiplication by tweaking the Montgomery Ladder. If we first do a conjugate addition, and then the addition with update, we follow the Montgomery structure where the coordinates get doubled and added each round, with the difference $R_1 - R_0 = P$, just as initially designed. Note that we do an initial doubling in Algorithm 9, to make sure that we never end up with the zero point, identical points or inverses in the algorithm, which would make the algorithm fail. Note that the initial doubling also change the point P so that the Jacobian points P' and $2P'$ share Z-coordinates.

Algorithm 9 Co-Z Montgomery Ladder

Input: Point P, scalar $k = (k_{n-1}, k_{n-2}, \ldots, k_1, k_0)$.
Output: Point $Q = [k]P$.

1: $(R_1, R_0) \leftarrow (2P', P')$ # InitialDouble
2: **for** $i = n - 2$ **downto** 1 **do**
3: $\quad (R_{1-k_i}, R_{k_i}) \leftarrow (R_{k_i} + R_{1-k_i}, R_{k_i} - R_{1-k_i})$ # XYCZ-ADDC
4: $\quad (R_{k_i}, R_{1-k_i}) \leftarrow (R_{k_i} + R_{1-k_i}, R_{1-k_i})$ # XYCZ-ADD
5: **end for**
6: $(R_{1-k_0}, R_{k_0}) \leftarrow (R_{k_0} + R_{1-k_0}, R_{k_0} - R_{1-k_0})$ # XYCZ-ADDC
7: $\lambda = Z^{-1}$ # FinalInvZ
8: $(R_{k_0}, R_{1-k_0}) \leftarrow (R_{k_0} + R_{1-k_0}, R_{1-k_0})$ # XYCZ-ADD
9: **return** $(X_0\lambda^2, Y_0\lambda^3)$

Also note that because XYCZ-ADDC calculate the sum and difference between the input, one of the resulting points will always be $\pm P$. Hence, we can make use of this in the last calculations in the algorithm to recover the Z-coordinate. Because we know how the Z-coordinate will change in the last calculation, we include this in the calculation of the final Z-coordinate.

We must note that because the addition formulas fail when one of the points are zero, or if the points are identical or inverses, the scalar multiplication algorithm fail if that ever happens. If the scalar is a number close to a multiple of the group order n, the scalar multiplication results in the zero point. Examples of values that makes the algorithm fail are $n-1$, $2n-2$ and $2n-1$. Additionally, because we perform an initial doubling before the loop, and also a final doubling and addition after the loop, we require that the scalar must be at least two bits large. The scalar of value 1 will be treated as the value 3, because it uses the single bit twice, as if there where two ones in the binary representation. These are keys that are very unlikely to occur; nevertheless, because of its intrinsic design, these keys make the algorithm fail.

8 Countermeasures

There is no way to ensure that all devices are secure against any side-channel attack against any implemented cryptographic algorithm. That said, making good decisions on how to implement the algorithms, and including reasonable countermeasures, can make it nigh infeasible for an attacker to learn any secret information. The literature about side channel attacks often gives feasible solutions for how to protect your systems without being forced to add too much code or making the program too time-consuming. This make it a lot harder for an attacker to complete a successful attack.

Countermeasures against side-channel attacks can be classified into two categories. In the first category, one tries to eliminate or minimize information leakage. This is achieved by reducing the signal-to-noise ratio of the side-channel signals. In the second category, one tries to ensure that the information that does leak through side-channels cannot be exploited to recover secrets. Typically, one will implement a combination of such countermeasures. We will in this section mention the most common countermeasures, explain which attack it prevents, and why they work that way.

For more information about side-channel attacks we refer the reader to "Algorithms, Key Size and Protocols Report" by Smart et al. [32], "Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing" by Zhou and Feng [39], and "State-of-the-art of secure ECC implementations: a survey on known side-channel attacks and countermeasures" by Fan et al. [14]. The following countermeasures are described in detail by Fan et al. [14] and Danger et al. [13].

8.1 Constant Time Code

Regularizing point additions. Regularizing the code and making sure that the additions and doublings are dependent on each other and performed for each bit of the key is one of the main ideas to defend against timing attacks and SPA. The classical double-and-add-algorithm leaks information about the bits in the secret key, as it branches its additions and multiplications depending on if the bit at each position is 1 or 0. We solve this by using the Montgomery Ladder.

Regularizing the key. Both timing attacks and SPA can be performed to learn the bit size of the key. The way to protect against the attacks is to regularize the key by making it some fixed size. For example, one can set the most significant bit to one, and let the remaining part of the key be random values. Otherwise, one may add a small multiple of the curve order n to the key before computing the scalar multiplication until the most significant bit of the key is set.

8.2 Randomized Projective Coordinates

Constant time and constant power consuming code does not necessarily defend a system against differential power analysis, due to dependence between consecutive calculations in the scalar multiplication algorithm. A effective countermeasure against DPA in ECC is to randomize the projective coordinates to make all traces different. This defends against statistical models trying to average out the noise.

Any projective point (X, Y, Z) on an elliptic curve represents the same affine point as the scaled projective point $k \cdot (X, Y, Z) = (k^n \cdot X, k^m \cdot Y, k \cdot Z)$ for any nonzero k, because $(x, y) = (X/(k \cdot Z)^n, Y/(k \cdot Z)^m)$ for all k. This give us the opportunity to multiply the projective base point $P = (x, y, 1)$ with a randomly chosen scalar k before we start our scalar multiplication. In the end, we get the same result when we go back to affine coordinates (x, y). Of course, this is only secure if we have a proper and safe way to generate random numbers, and a secure way of calculating the inverse of Z.

If we combine the regular Montgomery scalar multiplication version of the double-and-add-algorithm with randomized scalar multiplication, we efficiently defend against timing attacks, simple power analysis and differential power analysis. This is described in detail by Coron [12].

8.3 Scalar Blinding

Another countermeasure against DPA is scalar blinding, which means that we do our scalar multiplication with a new secret and random scalar every time instead of the fixed secret scalar. This way the attacker can't measure differences on repeatable computations with the same secret scalar, and hence, is not able to learn any secret information.

We blind a scalar d, by choosing a random scalar r and calculating $d' = d + r \cdot n$, where n is the order of the elliptic curve group. Then we calculate $[d']P = [d]P = Q$ to obtain the scalar multiple Q, because $[n]P = O$.

We must also be careful when calculating the inverses k^{-1} of k, but we can protect this operation using a similar trick. We first generate a new random number k', and do the following operations: $s = k \cdot k', t = 1/s, k^{-1} = k' \cdot t$. Another way to calculate multiplicative inverses is to use Fermat's little theorem $k^{p-1} \equiv 1 \mod p$ to calculate $k^{p-2} \equiv k^{-1}$. This is in general a bit slower; however, it doesn't require us to randomize the process.

8.4 Check for Correct Data

The last countermeasure we discuss is to check for correctness of inputs and outputs. In the literature, we can find series of attacks that take advantage of implemented algorithms which leak a lot of information because they do not check that all the data involved was correct. Zuccherato [40] recommends the following procedure to verify correct input and output:

1. Check that Q is not equal to the identity $\mathcal{O}$.
2. Check that both coordinates of Q are elements of the finite field F_p.
3. Check that Q is on the elliptic curve.
4. Check that $[n]Q = \mathcal{O}$ for n being the order of the elliptic curve group.

9 Libraries

In this section we present the libraries of comparison, based on a literature review in combination with code inspection. We document the elliptic curve point representations and scalar multiplication algorithms used in the libraries, in addition to the implemented countermeasures against SCA and fault attacks. We also note available hash functions, signing algorithms and additional features.

9.1 Micro-ECC

Micro-ECC [20] is an ECC library created for embedded devices. It is well documented that the library is fast and the code size is small, partly because of the implementations of fast number theoretic functions and inline Assembly optimizations (which we will not focus on in the paper). See Table 3 and Table 6 in Mossinger et al. [24] for an overview of Micro-ECC timings, code footprint and energy consumption. The author of Micro-ECC writes that the library is "resistant to known side-channel attacks", but the claims are not properly supported, neither in the literature nor in the documentation of the library.

Micro-ECC is mentioned in the literature several times. See Roy et al. [30], Mahé and Chauvet [21], Walz et al. [34], Clercq et al. [11] and Tausig and Schmidt [33], but all of these papers focus on the speed and code footprint of the library, while none expand our knowledge beyond the previously mentioned paper by

Mossinger et al. [24]. The only paper we can find that mentions the security in Micro-ECC is Nascimento et al. [25]. We quote the author's comment about Micro-ECC side-channel security from Table 1: "Not documented; apparently randomized projective coordinates".

By code inspection of Micro-ECC, we find that:

- Micro-ECC support NIST curves P-160, P-192, P-224, P-256 and P-384.
- Co-Z coordinates is used to represent the points of the elliptic curves. This representation allows for the fastest short Weierstrass curve additions.
- The co-Z Montgomery Ladder is used for scalar multiplication, as described in Algorithm 9 in Rivain [29]. Note that it uses Jacobian doubling in the start of the algorithm, instead of the doubling function initially described.
- The projective coordinates are randomized before computing the scalar multiplication while using the ECDH protocol, but not when computing a public key from a private key, and not when signing messages using ECDSA.
- The key is regularized before any scalar multiplication to be one bit larger than the elliptic curve group, as described by Brumley and Tuveri [9].
- Micro-ECC checks if the result of a scalar multiplication is the point of infinity, but does not check if the input or output points are on the curve.
- There is no dynamic memory allocation (in particular no `malloc()` nor `memcpy()`), no table look-ups and no branching on secret key material.
- Scalar blinding is used to hide the secret values when computing inverses.
- The library offer the option for deterministic signing of messages as described by Pornin [27], to avoid faulty randomness.
- The library does not provide built-in hash functions, and depends on external libraries to be able to sign messages. However, the signing algorithm is designed in a way so that it is easy to plug in any external hash functions.
- The library does misbehave for several boundary values when using the regularization of the key in combination with the co-Z scalar multiplication. The library does not check for invalid values 1, $n-2$ and $n-1$. The scalar multiplication results in the identify point for these values.

We conclude that Micro-ECC is secure against timing attacks and SPA, while only ECDH is secure against DPA. Micro-ECC is secure against some fault attacks since the curves are of prime order, and it checks that the result after scalar multiplication is not equal to the identity point. However, it does not by default verify that input points and output points in ECDH and ECDSA are on the curve.

9.2 Improved Micro-ECC

Micro-ECC is secure against most side-channel attacks, but its security could still be improved. Randomized projective coordinates are used some places, but not all. Functions for validating points are available, but not always used. The regularized key has failing boundary values for the scalar multiplication that should not have been allowed to be computed in the first place. It is easy to improve Micro-ECC. We include a version called Improved Micro-ECC in our final results.

- We add randomization to the scalar multiplication when computing the public key from the private key. We also check that the public key is valid.
- We verify that the received public key is valid before computing the shared secret. We also check that the output is a valid shared key.
- We add randomization to the scalar multiplication when computing the signature of a message. We also check that the curve point is valid.

9.3 Bear SSL

Bear SSL [28] is a full TLS library supporting TLS 1.0, 1.1 and 1.2, and includes RSA and symmetric cryptography in addition to the elliptic curve cryptography part that we are interested in. The library is written with embedded devices in mind, is very conservative in memory use, and implements primitives using big integers. We find that:

- Bear SSL support NIST curves P-256, P-384 and P-521, and Curve25519.
- Bear SSL uses Jacobean coordinates to represent points on the elliptic curve.
- Montgomery Ladder with a two-bit-window is used for scalar multiplication, to make only one add every two doublings, but requires more memory use. This is similar to Algorithm 11 in Rivain [29].
- The projective coordinates are randomized before any scalar multiplication.
- To avoid the leading-zero-bits timing attacks, Bear SSL sets the most-significant bit of the secret ephemeral scalar to 1.
- All finite-field operations have constant run-time.
- All points are validated by verifying the curve equation.
- The library offers the option for deterministic signing of messages.
- The library includes the hash functions MD5, SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512.

9.4 mbed TLS

mbed TLS [3] (formerly known as PolarSSL) is a full SSL/TLS library supporting SSL 3.0 and TLS 1.0, 1.1 and 1.2. The library is well documented and the code is easily readable. We summarize the main ECC features:

- mbed TLS uses projective coordinates to represent the elliptic curve points.
- The scalar multiplication is computed by using the Montgomery Ladder.
- The points are randomized before scalar multiplication.
- All input and output points are verified to belong to the elliptic curve.
- Scalar blinding is used to hide the secret values when computing inverses.
- The library includes the hash functions MD2, MD4, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 and RIPEMD-160.

9.5 FLECC

The Flexible Elliptic Curve Cryptography (FLECC) is written by Wenger et. al. [37]. See Table 4 in Wenger et al. [36] for an overview over FLECC timings, code footprint and energy consumption. By code inspection and literature review we report that:

- FLECC uses projective coordinates to represent points on the elliptic curve.
- Montgomery Ladder is used for computing scalar multiplications, as described by Hutter et al. [16].
- The projective coordinates are randomized before any scalar multiplication.
- To avoid the leading-zero-bits timing attacks, FLECC set the most-significant bit of the secret ephemeral scalar to 1, as documented in Wenger et al. [36].
- All finite-field operations have constant run-time.
- All input and output points are verified to belong to the elliptic curve.
- The library includes the hash functions SHA-1, SHA-224, and SHA-256.

10 Conclusion

We conclude all our results in table 1. The first SCA security analysis of Micro-ECC documents that the library is secure against timing attacks and SPA, in addition to most DPA and fault attacks. This dramatically improves upon the first impression given by Nascimento et. al. [25]. We easily improve the library to be secure against all mentioned attacks, making Improved Micro-ECC the fastest and most secure (together with FLECC and mbed TLS) ECC library available. FLECC and mbed TLS are also secure against all SCA mentioned, while Bear SSL has no countermeasures against DPA.

Table 1. Security of the ECC libraries, extending Nascimento et. al. [25]. We updated the security of Micro-ECC, added FLECC, mbed TLS and Improved Micro-ECC, in addition to update WolfSSL, RELIC and MIRACL based on a quick lookup of changes made since it was last documented. ✓means secure, ∃ means some countermeasures implemented, ✗means no countermeasures implemented, and - means not documented.

Library: || Security:	Timing	SPA	DPA	Verify Input	Verify Output
FLECC [37]	✓	✓	✓	✓	✓
mbed TLS [3]	✓	✓	✓	✓	✓
Improved Micro-ECC	✓	✓	✓	✓	✓
Micro-ECC [20]	✓	✓	∃	∃	∃
Bear SSL [28]	✓	✓	✗	✓	✓
NaCl [8]	✓	✓	✗	✓	-
WolfSSL [38]	∃	✗	✗	✓	✓
RELIC [2]	∃	✓	✗	-	-
MIRACL [10]	✗	✗	✗	✓	✗
CRS ECC [31]	✗	✗	✗	-	-
WM-ECC [35]	✗	✗	✗	-	-
TinyECC [19]	✗	✗	✗	-	-
AVR-Crypto-Lib [26]	✗	✗	✗	-	-
Wiselib [1]	✗	✗	✗	-	-

References

1. Amaxilatis, D.: A generic algorithms library for heterogeneous, distributed, embedded systems. https://github.com/ibr-alg/wiselib. Last accessed November 30, 2018.
2. Aranha, D., Gouvêa, C.: RELIC GitHub repository. https://github.com/relic-toolkit/relic. Last accessed November 30, 2018.
3. ARMmbed: mbed TLS GitHub repository. https://github.com/ARMmbed/mbedtls. Last accessed November 30, 2018.
4. Barker, E.: Digital Signature Standard (DSS). Federal Information Processing Standards Publication 186-4. (2013).
5. Barker, E. Chen, L., Roginsky, A., Vassilev, A., Davis, R.: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography. NIST Special Publication 800-56A Revision 3. (2018).
6. Barker, E., Roginsky, A.: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. In: NIST Special Publication (2015).
7. Bernstein, D., Lange, T.: Explicit-Formulas Database. http://www.hyperelliptic.org/EFD. Last accessed November 30, 2018.
8. Bernstein, D., Lange, T., Schwabe, P.: NaCl: Networking and Cryptography library. http://nacl.cr.yp.to/. Last accessed November 20 2018.
9. Brumley, B., Tuveri, N.: Remote Timing Attacks are Still Practical. In: Cryptology ePrint Archive, Report 2011/232 (2011).
10. Certivox: MIRACL GitHub repository. https://github.com/miracl/MIRACL. Last accessed November 30, 2018.
11. Clercq, R., Uhsadel, L., Herrewege, A., Verbauwhede, I.: Ultra Low-Power implementation of ECC on the ARM Cortex-M0+. In: Cryptology ePrint Archive, Report 2013/609 (2013).
12. Coron, J.: Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In: Cryptographic Hardware and Embedded Systems. CHES 1999. Lecture Notes in Computer Science, vol 1717. Springer (1999).
13. Danger, J., Guilley, S., Hoogvorst, P., Murdica, C., Naccache, D.: A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards. In: Journal of Cryptographic Engineering 3:241. Springer (2013).
14. Fan, J., Guo, X., Mulder, E., Schaumont, P., Preneel, B. Verbauwhede, I.: State-of-the-art of secure ECC implementations: a survey on known side-channel attacks and countermeasures. In: 3rd IEEE International Symposium on Hardware-Oriented Security and Trust - HOST 2010. IEEE (2010).
15. Goundar, R., Joye, M., Miyaji, A.: Co-Z Addition Formulæ and Binary Ladders on Elliptic Curves. In: Cryptology ePrint Archive, Report 2010/309. (2010).
16. Hutter, M., Joye, M., Sierra, Y.: Memory-Constrained Implementations of Elliptic Curve Cryptography in Co-Z Coordinate Representation. In: Progress in Cryptology - AFRICACRYPT 2011, Lecture Notes in Computer Science. Springer (2011).
17. Kocher, P.: Cryptanalysis of Diffie-Hellman, RSA, DSS, and Other Systems Using Timing Attacks. In: Advances in Cryptology, CRYPTO '95: 15TH Annual International Cryptology Conference (1995).
18. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Advances in Cryptology – CRYPTO' 99. Springer (1999).
19. Liu, A., Ning, P.: TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. Last accessed November 30, 2018. http://discovery.csc.ncsu. edu/software/TinyECC/ver1.0/index.html.

20. MacKay, K.: Micro-ECC GitHub repository.
https://github.com/kmackay/microecc. Last accessed November 30, 2018.
21. Mahé, E., Chauvet, J.: Fast GPGPU-Based Elliptic Curve Scalar Multiplication. In: Cryptology ePrint Archive, Report 2014/198 (2014).
22. Meloni, N.: New Point Addition Formulae for ECC Applications. In C.Carletand, B.Sunar, Arithmetic of Finite Fields, pages 189–201. Springer (2007).
23. Montgomery, P.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation, 48:243–264. (1987)
24. Mossinger, M., Petschkuhn, B., Bauer, J., Staudemeyer, R., Wojcik, M., Pohls, H.: Towards quantifying the cost of a secure IoT: Overhead and energy consumption of ECC signatures on an ARM-based device. In: IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (2016).
25. Nascimento, E., Chmielewski, L., Oswald, D., Schwabe, P.: Attacking embedded ECC implementations through cmov side channels. In: Selected Areas in Cryptology 2016, Lecture Notes in Computer Science 10532, Springer (2017).
26. Otte, D.: Avr-crypto-lib.
https://git.cryptolib.org/avr-crypto-lib.git. Last accessed November 30, 2018.
27. Pornin, T.: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). In: Network Working Group, RFC6979. The Internet Society (2013).
28. Pornin, T.: Bear SSL Gitweb repository.
https://www.bearssl.org/gitweb/?p=BearSSL. Last accessed November 30, 2018.
29. Rivain, M.: Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves. In: Cryptology ePrint Archive, Report 2011/338 (2011).
30. Roy, D., Das, P., Mukhopadhyay, D.: ECC on Your Fingertips: A Single Instruction Approach for Lightweight ECC Design in GF(p). In: Cryptology ePrint Archive, Report 2015/1225 (2015).
31. Sigma: ECDSA and ECDH cryptographic algorithms for 8-bit AVR microcontrollers. http://www.cmmsigma.eu/products/crypto/index.en.html.
32. Smart, N. et al.: Algorithms, Key Size and Protocols Report (2018). In: ECRYPT -– CSA, H2020-ICT-2014 — Project 645421 (2018).
33. Tausig, M., Schmidt, S.: Performance Evaluation of Cryptographic Operations on a SAMR21-XPRO Board. In: RIOT-OS Summit Berlin (2016).
34. Walz, A., Kehret, O., Sikora, A.: Comparison of cryptographic implementations for embedded TLS. In: "Embedded World Conference, Nurnberg (2016).
35. Wang, H.: WM-ECC is an Elliptic Curve Cryptography primitive suite developed exclusively for wireless sensor motes.
http://cis.csuohio.edu/ hwang/WMECC.html. Last accessed November 30, 2018.
36. Wenger, E., Unterluggauer, T., Werner, M.: 8/16/32 Shades of Elliptic Curve Cryptography on Embedded Processors. In: Progress in Cryptology – INDOCRYPT 2013, Springer (2013).
37. Wenger, E., Unterluggauer, T., Werner, M.: FLECC GitHub repository.
https://github.com/IAIK/flecc_in_c. Last accessed November 30, 2018.
38. Wolf SSL.: Wolf SSL GitHub repository.
https://github.com/wolfSSL/wolfssl. Last accessed November 30, 2018.
39. Zhou, Y., Feng, D.: Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. In: Cryptology ePrint Archive, Report 2005/388. (2005).
40. Zuccherato, R.: Methods for Avoiding the "Small-Subgroup" Attacks on the Diffie-Hellman Key Agreement Method for S/MIME. In: Network Working Group, RFC2785. The Internet Society (2000).