Analysis and Implementation of ECDSA over secp256r1 in `libecc`

Your Name
Cryptography Lab, XYZ University

June 8, 2025

Abstract

In this report, we analyze the ECDSA (Elliptic Curve Digital Signature Algorithm) implementation in the `libecc` library, focusing on the secp256r1 (P-256) curve. We cross-reference the FIPS 186-5 specification with the source code, detailing how domain parameters are loaded, how big-integer arithmetic is realized, how point-operations are implemented, and how the `ecdsa_sign()` and `ecdsa_verify()` functions faithfully follow the standard. Security considerations and testing results are also discussed.

Contents

1 Introduction

1.1 Background

ECDSA is an elliptic-curve variant of the Digital Signature Algorithm, widely used for secure digital signatures. secp256r1 (P-256) is a NIST-recommended curve with 256-bit security.

1.2 Purpose and Scope

This report details the mapping between the ECDSA specification (FIPS 186-5) and the `libecc` C implementation, limited to secp256r1. We assume familiarity with basic modular arithmetic and elliptic-curve fundamentals.

2 ECDSA Specification Overview

2.1 Domain Parameters for secp256r1

The secp256r1 curve is defined by:

$$\begin{aligned}
p &= \mathrm{0xFFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF},\\
a &= p - 3,\\
b &= \mathrm{0x5AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B},\\
G_x &= \mathrm{0x6B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296},\\
G_y &= \mathrm{0x4FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5},\\
n &= \mathrm{0xFFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551},\\
h &= 1.
\end{aligned}$$

2.2 Key-Pair Generation

1. Select private key $d \in [1, n-1]$ uniformly at random.
2. Compute public key $Q = d \cdot G$ on the curve.

2.3 Signature Generation

Steps per FIPS 186-5:

1. Compute $e = \mathrm{Hash}(m) \bmod n$.
2. Select random nonce $k \in [1, n-1]$.
3. Compute $(x_1, y_1) = k \cdot G$, set $r = x_1 \bmod n$; if $r = 0$, go back to 2.
4. Compute $s = k^{-1}(e + d\,r) \bmod n$; if $s = 0$, go back to 2.
5. Output signature (r, s).

2.4 Signature Verification

1. Verify $1 \le r, s \le n - 1$. If not, reject.
2. Compute $e = \mathrm{Hash}(m) \bmod n$.
3. Compute $w = s^{-1} \bmod n$.
4. Compute $u_1 = e\,w \bmod n$, $u_2 = r\,w \bmod n$.
5. Compute $P = u_1 \cdot G + u_2 \cdot Q$. If P is at infinity, reject.
6. Let $v = P_x \bmod n$. Accept if $v = r$.

3 Code Repository Layout

```
libecc/
├── include/libecc
│   ├── curves/known/ec_params_secp256r1.h
│   ├── curves/ec_params.h
│   ├── curves/aff_pt.h
│   ├── curves/ec_shortw.h
│   ├── nn/nn.h
│   ├── nn/nn_rand.h
│   ├── sig/ecdsa_common.h
│   ├── sig/ecdsa.h
│   ├── hash/sha256.h
│   └── utils/utils_rand.h
└── src
├── curves/ec_params.c
├── curves/aff_pt.c
├── curves/ec_shortw.c
├── nn/nn.c, nn_add.c, nn_mul.c, nn_modinv.c, nn_rand.c
├── sig/ecdsa_common.c, ecdsa.c
├── hash/sha256.c
└── utils/utils_rand.c
```

4 Mapping Specification to Code

4.1 Domain Parameters

4.1.1 Code: ec_params_secp256r1.h

```
#define SECP256R1_P_HEX "
    FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFF"
#define SECP256R1_A_HEX "
    FFFFFFFF00000001000000000000000000000000FFFFFFFFFFFFFFFFFFFFFFFC"
#define SECP256R1_B_HEX "5
    AC635D8AA3A93E7B3EBBD55769886BC651D06B0CC53B0F63BCE3C3E27D2604B"
```

```
#define SECP256R1_GX_HEX "6
    B17D1F2E12C4247F8BCE6E563A440F277037D812DEB33A0F4A13945D898C296"
#define SECP256R1_GY_HEX "4
    FE342E2FE1A7F9B8EE7EB4A7C0F9E162BCE33576B315ECECBB6406837BF51F5"
#define SECP256R1_N_HEX "
    FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551"
#define SECP256R1_H 1
```

4.1.2 Code: ec_params_load_shortw

```
int ec_params_load_shortw(ec_curve_shortw_t *C, const char *name) {
        if (strcmp(name, "SECP256R1") == 0) {
                nn_set_hex(&C->p, SECP256R1_P_HEX);
                nn_set_hex(&C->a, SECP256R1_A_HEX);
                nn_set_hex(&C->b, SECP256R1_B_HEX);
                nn_set_hex(&C->Gx, SECP256R1_GX_HEX);
                nn_set_hex(&C->Gy, SECP256R1_GY_HEX);
                nn_set_hex(&C->n, SECP256R1_N_HEX);
                C->h = SECP256R1_H;
                return 0;
        }
        return -1;
}
```

4.2 Big-Integer (NN) Arithmetic

```
// nn_t definition in nn.h
typedef struct {
        uint32_t words[NN_MAX_WORDS];
        int used;
        int sign;
} nn_t;

// Core operations:
void nn_set_hex(nn_t *r, const char *hexstr);
void nn_mod_mul(nn_t *r, const nn_t *a, const nn_t *b, const nn_t *m);
void nn_mod_inv(nn_t *r, const nn_t *a, const nn_t *m);
```

4.3 Elliptic-Curve Arithmetic

```
// ec_shortw_add in ec_shortw.c
void ec_shortw_add(affine_point_t *R,
const affine_point_t *P,
const affine_point_t *Q,
const ec_curve_shortw_t *C) {
        // handle infinity cases...
        nn_sub(&num, &Q->y, &P->y);
```

```
        nn_sub(&den, &Q->x, &P->x);
        nn_mod_inv(&den_inv, &den, &C->p);
        nn_mod_mul(&lambda, &num, &den_inv, &C->p);
        // compute x3, y3...
        R->x = x3; R->y = y3; R->is_inf = 0;
}
```

4.4 ECDSA Key Generation

```
int ecdsa_keygen(ssize_t curve_id, nn_t *priv, affine_point_t *pub) {
        ec_curve_shortw_t C;
        get_curve_shortw(&C, (int)curve_id);
        do { nn_rand(priv, &C.n); }
        while (nn_is_zero(priv) || nn_cmp(priv, &C.n) >= 0);
        affine_point_t G = { C.Gx, C.Gy, 0 };
        ec_shortw_mul(pub, &G, priv, &C);
        return 0;
}
```

4.5 ECDSA Signature Generation

```
int ecdsa_sign(ssize_t curve_id, const nn_t *priv,
const uint8_t *msg, size_t msglen,
ecdsa_sig_t *sig) {
        ec_curve_shortw_t C;
        get_curve_shortw(&C, (int)curve_id);
        uint8_t hash[32]; sha256(msg,msglen,hash);
        nn_t e; ecdsa_hash_to_int(&e,hash,&C.n);
        nn_t k, k_inv, tmp; affine_point_t Rpt, G = {C.Gx,C.Gy,0};
        do {
                do { nn_rand(&k,&C.n); } while (...);
                ec_shortw_mul(&Rpt,&G,&k,&C);
                nn_mod(&sig->r,&Rpt.x,&C.n);
                if (nn_is_zero(&sig->r)) continue;
                nn_mul(&tmp, priv, &sig->r);
                nn_add(&tmp, &tmp, &e);
                nn_mod(&tmp, &tmp, &C.n);
                nn_mod_inv(&k_inv, &k, &C.n);
                nn_mod_mul(&sig->s, &k_inv, &tmp, &C.n);
        } while (nn_is_zero(&sig->s));
        return 0;
}
```

4.6 ECDSA Signature Verification

```
int ecdsa_verify(ssize_t curve_id,
const affine_point_t *pub,
```

```
const uint8_t *msg, size_t msglen,
const ecdsa_sig_t *sig) {
        ec_curve_shortw_t C; get_curve_shortw(&C,(int)curve_id);
        if (invalid_range(sig->r, sig->s, &C.n)) return 0;
        uint8_t hash[32]; sha256(msg,msglen,hash);
        nn_t e; ecdsa_hash_to_int(&e,hash,&C.n);
        nn_t w; nn_mod_inv(&w, &sig->s, &C.n);
        nn_t u1,u2; nn_mod_mul(&u1,&e,&w,&C.n); nn_mod_mul(&u2,&sig->r,&w,&C.n)
            ;
        affine_point_t G={C.Gx,C.Gy,0}, u1G,u2Q, Rpt;
        ec_shortw_mul(&u1G,&G,&u1,&C);
        ec_shortw_mul(&u2Q,pub,&u2,&C);
        ec_shortw_add(&Rpt,&u1G,&u2Q,&C);
        if (Rpt.is_inf) return 0;
        nn_t v; nn_mod(&v,&Rpt.x,&C.n);
        return (nn_cmp(&v,&sig->r)==0);
}
```

5 Testing and Validation

5.1 Self-Test Vectors

Discuss test harness in `src/tests/ec_self_tests.c` using known signatures.

5.2 Randomized and Interoperability Tests

Describe cross-verification with OpenSSL, random message tests, etc.

6 Security Considerations

Discuss constant-time, public-key validation, deterministic k (RFC 6979), side-channels.

7 Conclusion

Summarize mapping fidelity, correctness, compliance, and recommended improvements.

References

- FIPS 186-5: Digital Signature Standard (DSS).
- SEC 2: Recommended Elliptic Curve Domain Parameters.
- NIST P-256 specification.
- `libecc` source repository (commit XYZ).