The Big Data Processing using Elliptic Curve Cryptography

A Report on the Design and Implementation of
Secure File and Tamper-Evident Log Processing Utilities

Ji, Yong-hyeon
M.S. Student
`hacker3740@kookmin.ac.kr`

Department of Cyber Security
Kookmin University

June 11, 2025

Contents

Chapter 1

Introduction

This document details the design, implementation, and analysis of two C applications that leverage Elliptic Curve Digital Signature Algorithm (ECDSA) to ensure data integrity and non-repudiation in big data processing environments.

The first application, '`filecheck_ecc`', provides command-line-based file signing and verification. The second application, a '`logtool_ecc`' and '`logaggregator_ecc`', establishes a framework for creating cryptographically signed, tamper-evident log streams suitable for distributed systems. Both systems are built upon the modular `libecc`[BEF17] cryptographic library, utilizing the SECP256R1 curve and SHA-256 (or SHA3-256) hashing algorithm.

This document presents the system architecture, a comprehensive developer API guide, performance benchmarks, and security considerations for both applications, demonstrating a practical approach to embedding strong cryptographic guarantees into data-intensive workflows.

1.1 Project Scope and Objectives

This report presents two prototype applications built around ECDSA on the NIST P-256 curve:

1. A *File Integrity Checker* that signs and verifies large files via streaming SHA-256 and chunked ECDSA operations.

2. A *Tamper-Evident Log Aggregator* for real-time signing of high-volume log entries with rotating output chunks.

The primary objectives are to demonstrate scalable ECDSA integration, evaluate performance trade-offs, and document design patterns for secure big data processing.

This project aims to address the aforementioned problem by designing and implementing a framework of two C-based applications built upon the `libecc` library. The scope is focused on providing practical, command-line-driven tools that can be easily scripted and integrated into existing data pipelines. The primary objectives are as follows:

1. **Develop a Secure File Integrity Tool:** To create a robust utility, `filecheck_secure`, for signing and verifying the integrity and authenticity of large data files, such as datasets, archives, and backups.

2. **Develop a Secure Logging Framework:** To build a suite of applications, `logtool` and `logaggregator`, capable of generating cryptographically signed, tamper-evident log entries and processing them in a streaming fashion.

3. **Utilize Industry-Standard Cryptography:** To implement all functionalities using the Elliptic Curve Digital Signature Algorithm (ECDSA) with the widely adopted SECP256R1 (NIST P-256) curve and the SHA-256 hash algorithm.

4. **Provide a Clear Developer Guide:** To document the core API functions used from the underlying cryptographic library, enabling other developers to extend the tools or build new secure applications.

1.2 Summary of Contributions

- **Modular C implementations** of streaming ECDSA signing and verification, optimized for large files and logs.
- **Design patterns** for chunked hashing, memory-bounded processing, and rotation of signed data segments.
- **Performance evaluation** and best practices for deploying ECDSA in high-throughput Linux environments.

1.3 Report Structure

The remainder of this report is organized as follows:

- **Section 2:** Standards & Specifications (FIPS 186-5 ECDSA, P-256 parameters).
- **Section 3:** System Architecture and Design Patterns.
- **Section 4:** File Integrity Checker—implementation details and benchmarks.
- **Section 5:** Tamper-Evident Log Aggregator—pipeline design and throughput results.
- **Section 6:** Performance & Security Considerations, including key management.
- **Section 7:** Conclusion and Future Work.

1.3.1 Report Structure

The remainder of this report is organized as follows. Section 2 provides background on the cryptographic principles underlying the project, including ECC, ECDSA, and hash functions. Section 3 details the high-level system architecture and design choices. Sections 4 and 5 present the specific implementation details of the `filecheck_secure` tool and the secure logging suite, respectively. Section 6 offers a comprehensive developer's guide to the API and its usage. Section 7 provides a performance and security analysis of the implemented systems. Finally, Section 8 concludes the report with a summary of findings and suggestions for future work.

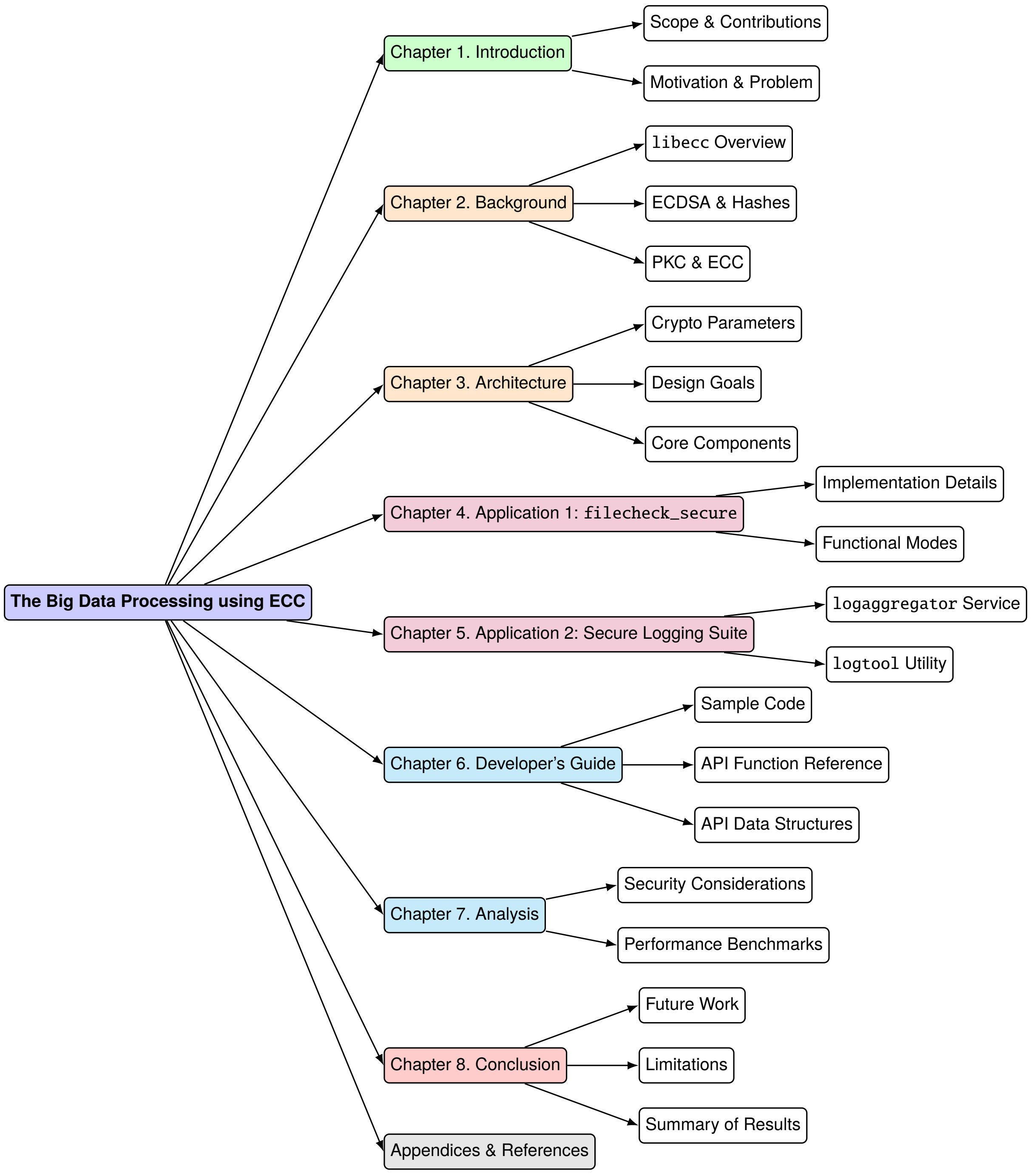


Figure 1.1: A hierarchical overview of the structure of document.

Chapter 2

Background and Preliminaries

2.1 Fundamentals of Public Key Cryptography

Public key cryptography relies on mathematical problems that are easy to compute in one direction but hard to invert without special knowledge (the private key). Common primitives include RSA, Diffie–Hellman, and elliptic-curve based schemes.

2.2 Elliptic Curve Cryptography (ECC)

An elliptic curve over a finite field $\mathbb{F}_p$ is defined by the Weierstrass equation:

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

where $a, b \in \mathbb{F}_p$ and the discriminant condition

$$4a^3 + 27b^2 \not\equiv 0 \pmod{p}$$

ensures no singularities.

2.3 The Elliptic Curve Digital Signature Algorithm (ECDSA)

2.3.1 Key Generation

A private key d is a randomly selected integer in $[1, n-1]$, where n is the order of the generator point G. The public key Q is computed as

$$Q = d\,G.$$

2.3.2 Signature Generation

To sign a message hash $e = H(m)$:

1. Select a random integer $k \in [1, n-1]$.
2. Compute the point $(x_1, y_1) = k\,G$.
3. Compute $r = x_1 \bmod n$. If $r = 0$, restart with a new k.
4. Compute $s = k^{-1}(e + r\,d) \bmod n$. If $s = 0$, restart with a new k.

The signature is the pair (r, s).

2.3.3 Signature Verification

To verify a signature (r, s) on a message hash e:

1. Verify that $r, s \in [1, n-1]$.
2. Compute $w = s^{-1} \bmod n$.
3. Compute
$$u_1 = e\,w \bmod n, \qquad u_2 = r\,w \bmod n.$$
4. Compute the point $(x_0, y_0) = u_1\,G + u_2\,Q$.
5. The signature is valid if
$$x_0 \bmod n \;\equiv\; r.$$

2.4 Hash Functions and their Role in Digital Signatures

Hash functions (e.g., SHA-256) produce fixed-size digests of arbitrary-length messages, ensuring both integrity and efficiency when signing large data by signing the digest instead of the full message.

2.5 `libecc` Cryptographic Library: An Overview

The `libecc` library provides optimized implementations of elliptic-curve primitives, including domain parameter management, key generation, signature algorithms, and secure random number generation, all conforming to industry standards like FIPS 186-5.

Chapter 3

System Architecture and Design

This chapter outlines the high-level design of the framework, detailing its core components, the technology stack, the primary design goals that guided development, and the rationale for the specific cryptographic parameters selected.

3.1 Core Components

The system is designed as a layered architecture to promote modularity and a clear separation of concerns. It consists of three primary logical layers: the Cryptographic Primitives Layer, the Application Logic Layer, and the Data I/O and Interface Layer. This structure, illustrated in Figure 4.1, ensures that the complex cryptographic operations are neatly encapsulated and abstracted away from the higher-level application functionality.

- **Cryptographic Primitives Layer:** This is the lowest layer of the stack, provided entirely by the `libecc` library. It is responsible for all fundamental cryptographic operations, including elliptic curve arithmetic, key pair generation, hash computation (SHA-256), and the core ECDSA signing and verification algorithms. This layer abstracts the complex mathematics of cryptography into a stable, verifiable API.
- **Application Logic Layer:** This intermediate layer contains the core logic for the `filecheck_secure` and secure logging applications. It orchestrates the cryptographic operations exposed by the layer below to achieve its functional goals. For example, the 'do_sign' function in `filecheck_secure` coordinates the process of hashing a file and then passing the resulting digest to the `libecc` signing function.
- **Data I/O and Interface Layer:** This is the highest layer, responsible for all interactions with the outside world. This includes parsing command-line arguments, reading and writing files from the filesystem (keys, signatures, and data files), and processing standard input/output streams for the logging suite. It ensures that data is correctly formatted and passed to the application logic layer.

3.2 Technology Stack

The selection of technologies was driven by the core requirements of performance, portability, and low-level system control.

- **Programming Language: C (C99 Standard):** The C language was chosen for its performance, minimal runtime overhead, and direct memory management capabilities, which

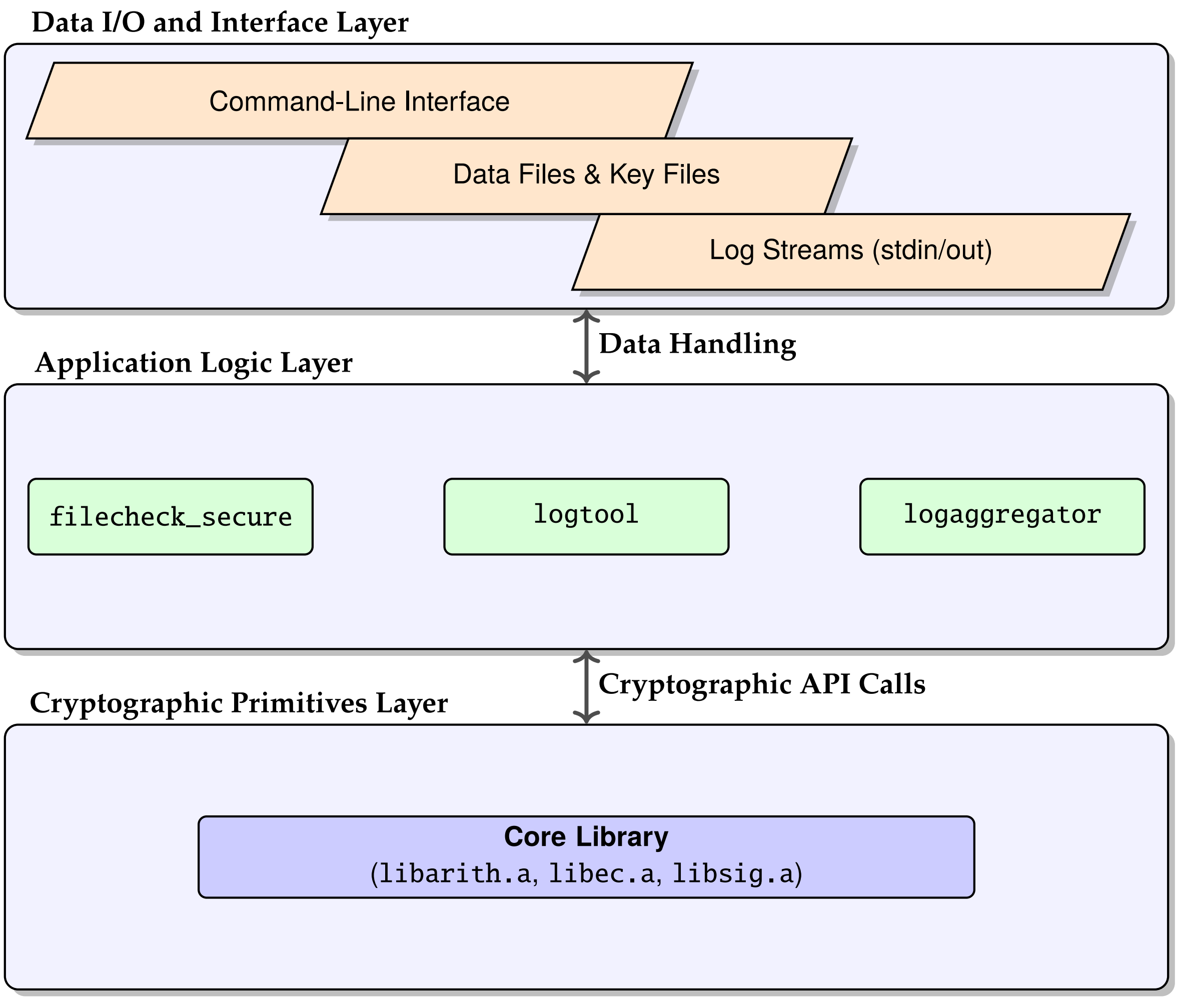


Figure 3.1: **System Architecture.** The framework is stratified into three distinct layers, ensuring a separation of concerns between user interaction, application business logic, and low-level cryptographic operations.

are critical for high-throughput data processing. Its ubiquity ensures maximum portability across POSIX-compliant operating systems commonly used in big data environments, such as Linux.

- **Cryptographic Library: `libecc`:** As detailed in the previous section, `libecc` provides the necessary cryptographic functions in a modular and configurable C-based package.
- **Build System: GCC and Make:** The standard GNU Compiler Collection (GCC) and Make are used for compilation and dependency management, ensuring a simple and universally understood build process.

3.3 Design Goals

The design of both applications was guided by three fundamental principles.

3.3.1 Security and Robustness

Security is the paramount concern. This was achieved by exclusively using well-vetted, industry-standard cryptographic algorithms and parameters. The implementation avoids creating any

Figure Placeholder: System Architecture Diagram

This diagram would show the three layers: 1. Data/Interface Layer (CLI, Files, Logs) at the top. 2. Application Logic Layer ('filecheck_secure', 'logtool', etc.) in the middle. 3. Cryptographic Primitives Layer ('libecc') at the bottom.

Figure 3.2: Layered System Architecture.

custom cryptographic primitives ("rolling your own crypto") and instead relies entirely on the `libecc` library. Robustness is addressed through comprehensive error handling at every stage, from command-line parsing to file I/O and cryptographic operations, ensuring that the applications fail safely and provide clear diagnostic messages.

3.3.2 Performance and Scalability

To be viable in big data contexts, the tools must be highly performant. The use of compiled C code minimizes execution overhead. For the `filecheck_secure` application, a critical design choice was the implementation of a streaming hash mechanism. The application reads and hashes large files in small, manageable chunks (e.g., 4096 bytes) rather than loading the entire file into memory. This approach ensures that the tools can process files of virtually any size with a minimal and constant memory footprint, making them highly scalable.

3.3.3 Modularity and Usability

The system is designed to be both modular and easy to use. The layered architecture separates cryptographic logic from application logic, making the code easier to maintain and audit. From a user's perspective, both applications are exposed as simple, self-contained command-line utilities. This design choice makes them trivial to integrate into existing shell scripts, cron jobs, or automated data processing workflows (e.g., Apache Airflow, shell-based ETL scripts), providing a low barrier to adoption for data engineering teams.

3.4 Cryptographic Parameter Selection

The choice of cryptographic parameters is fundamental to the security of the entire system. The selected parameters represent a modern, conservative choice that balances strong security with high performance.

3.4.1 Curve: NIST P-256 (SECP256R1)

The SECP256R1 elliptic curve, also known as NIST P-256 or prime256v1, was selected for this project. It is one of the curves standardized by the National Institute of Standards and Technology (NIST) in FIPS 186-4 and is the most widely used curve for commercial applications. It offers a 128-bit security level, which is considered secure against all known classical and quantum attacks for the foreseeable future. Its widespread adoption means it has been subject to extensive public scrutiny and is highly optimized in most cryptographic libraries, including `libecc`.

3.4.2 Hash Algorithm: SHA-256

The Secure Hash Algorithm 2 (SHA-2) with a 256-bit digest was chosen as the companion hash function. SHA-256 provides a security level that is commensurate with the 128-bit security of the P-256 curve. It is a FIPS-validated standard and remains a trusted workhorse for digital signatures, message authentication codes, and other cryptographic protocols. The streaming interface provided by `libecc` for SHA-256 was a key factor in its selection, enabling the efficient processing of large files as required by the project's design goals.

Chapter 4

Cryptographic Primitives

In this chapter, we examine how cryptographic primitives are integrated and how the abstraction layer orchestrates with key generation, hashing, and digital signature operations.

```
CryptoModule/
├── include
│   └── crypto
│       ├── cryptoarith.h
│       ├── crypto_ecc_config.h
│       ├── crypto_ecc_types.h
│       ├── cryptoec.h
│       ├── cryptosig.h
│       ├── curves
│       │   ├── aff_pt.h
│       │   ├── curves.h
│       │   ├── curves_list.h
│       │   ├── ec_params_external.h
│       │   ├── ec_params.h
│       │   ├── ec_params_secp256r1.h
│       │   ├── ec_shortw.h
│       │   └── prj_pt.h
│       ├── external_deps
│       │   ├── print.h
│       │   ├── rand.h
│       │   └── time.h
│       ├── fp
│       │   ├── fp_add.h
│       │   ├── fp_config.h
│       │   ├── fp.h
│       │   ├── fp_montgomery.h
│       │   ├── fp_mul.h
│       │   ├── fp_mul_redc1.h
│       │   ├── fp_pow.h
│       │   ├── fp_rand.h
│       │   └── fp_sqrt.h
│       ├── hash
│       │   ├── hash_algs.h
│       │   ├── hmac.h
│       │   ├── keccak.h
│       │   ├── sha256.h
│       │   ├── sha2.h
│       │   ├── sha3-256.h
│       │   ├── sha3.h
│       │   ├── shake256.h
│       │   └── shake.h
│       ├── nn
│       │   ├── nn_add.h
│       │   ├── nn_config.h
│       │   ├── nn_div.h
│       │   ├── nn_div_public.h
│       │   ├── nn.h
│       │   ├── nn_logical.h
│       │   ├── nn_modinv.h
│       │   ├── nn_mod_pow.h
│       │   ├── nn_mul.h
│       │   ├── nn_mul_public.h
│       │   ├── nn_mul_redc1.h
│       │   └── nn_rand.h
│       ├── sig
│       │   ├── ecdsa_common.h
│       │   ├── ecdsa.h
│       │   ├── ec_key.h
│       │   ├── sig_algs.h
│       │   └── sig_algs_internal.h
│       ├── utils
│       │   ├── ...
│       └── words
│       │   ├── ..
```

4.1 Overview of Libraries

A key feature of the library is its use of an abstraction layer to support multiple cryptographic algorithms through a unified interface.

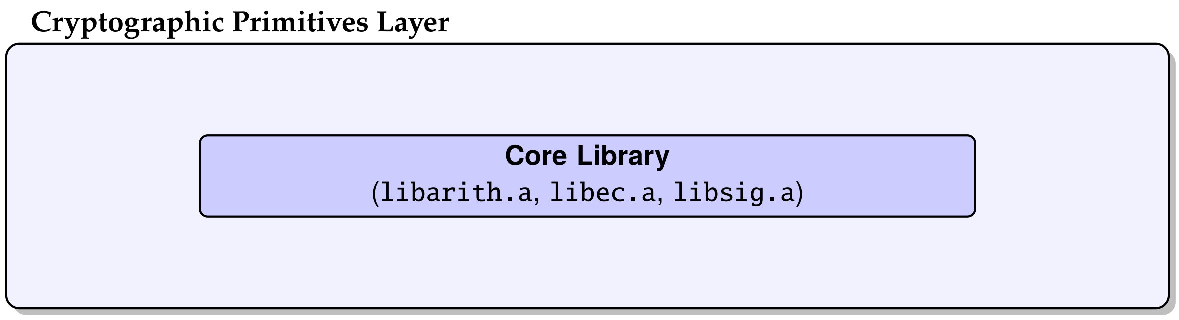

Cryptographic Primitives Layer
Core Library
(libarith.a, libec.a, libsig.a)

Figure 4.1: **Cryptographic Primitive Layer.** This is a low-level cryptographic operations includes three libraries: `libarith.a`, `libec.a`, `libsig.a`.

libarith.a

Code 4.1: include/cryptoarith.h

```c
#ifndef __CRYPTOARITH_H__
#define __CRYPTOARITH_H__
/* NN layer includes */
#include <crypto/nn/nn.h>
#include <crypto/nn/nn_logical.h>
#include <crypto/nn/nn_add.h>
#include <crypto/nn/nn_mul_public.h>
#include <crypto/nn/nn_mul_redc1.h>
#include <crypto/nn/nn_div_public.h>
#include <crypto/nn/nn_modinv.h>
#include <crypto/nn/nn_mod_pow.h>
#include <crypto/nn/nn_rand.h>
#include <crypto/utils/print_nn.h>
/* Fp layer include */
#include <crypto/fp/fp.h>
#include <crypto/fp/fp_add.h>
#include <crypto/fp/fp_montgomery.h>
#include <crypto/fp/fp_mul.h>
#include <crypto/fp/fp_sqrt.h>
#include <crypto/fp/fp_pow.h>
#include <crypto/fp/fp_rand.h>
#include <crypto/utils/print_fp.h>
```

libec.a

Code 4.2: include/cryptoec.h

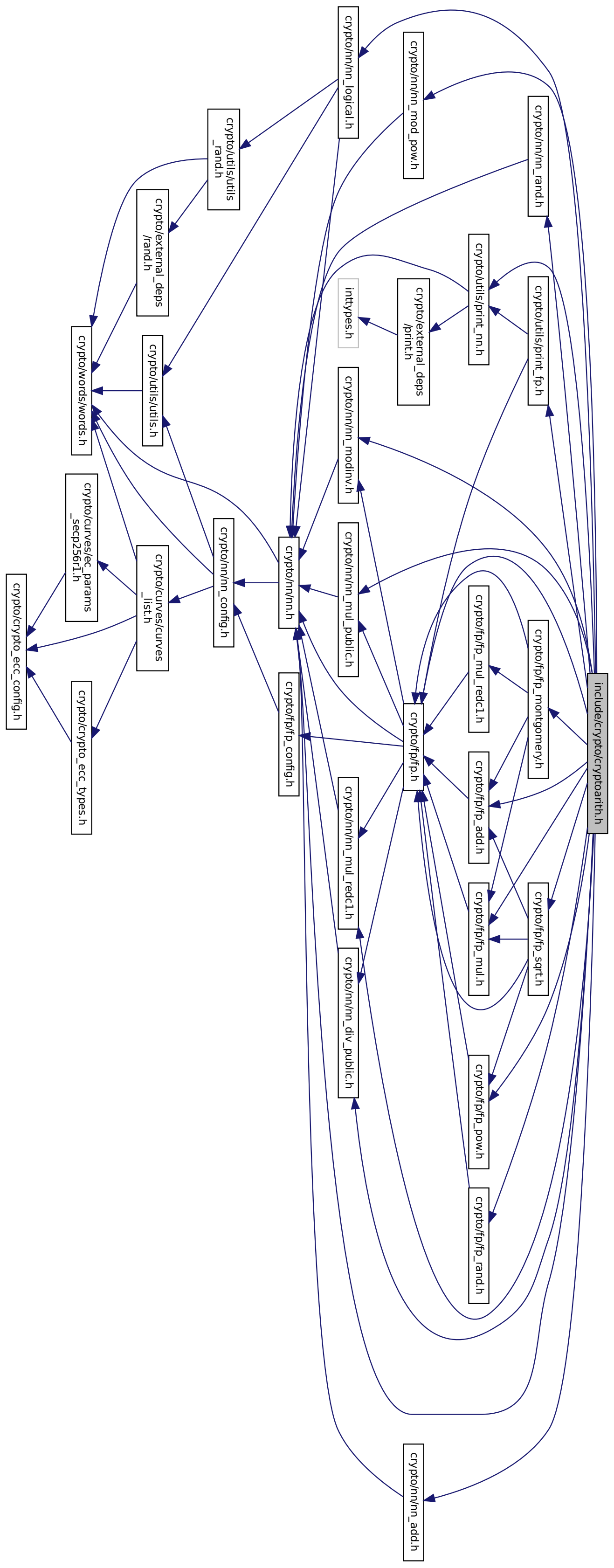
```c
#ifndef __CRYPTOEC_H__
#define __CRYPTOEC_H__
/* Include the cryptoarith package */
#include <crypto/cryptoarith.h>
/* Curve layer includes */
#include <crypto/curves/curves.h>
#include <crypto/curves/curves_list.h>
#include <crypto/curves/ec_shortw.h>
#include <crypto/curves/prj_pt.h>
#include <crypto/curves/aff_pt.h>
#include <crypto/utils/print_curves.h>
#endif /* __CRYPTOEC_H__ */
```

libsig.a

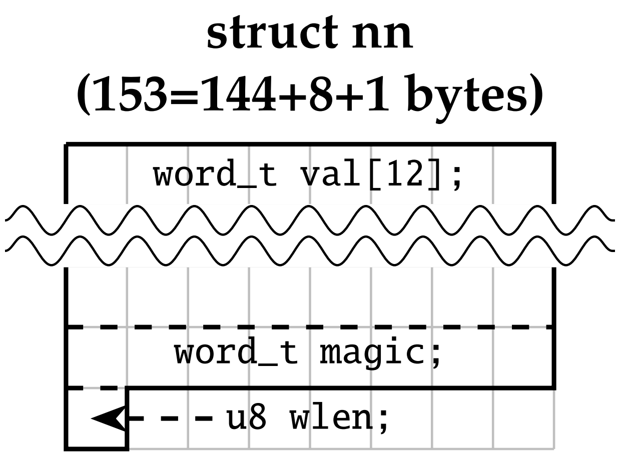
Code 4.3: include/cryptosig.h

```c
#ifndef __CRYPTOSIG_H__
#define __CRYPTOSIG_H__
/* Include the Elliptic Curves layer */
#include <crypto/cryptoec.h>
#include <crypto/crypto_ecc_config.h>
#include <crypto/crypto_ecc_types.h>
#include <crypto/sig/sig_algs.h>
#include <crypto/sig/ec_key.h>
#include <crypto/utils/dbg_sig.h>
/* Include the hash functions */
#include <crypto/hash/hash_algs.h>
/* Include the hmac functions */
#include <crypto/hash/hmac.h>
#endif /* __CRYPTOSIG_H__ */
```

4.2 Modular Arithmetic and Finite Field Library



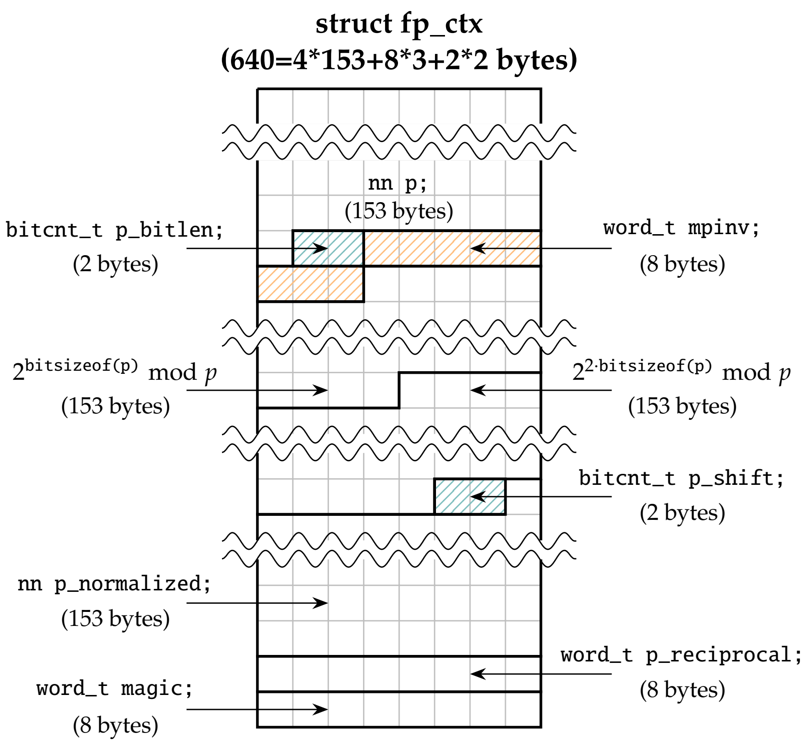
4.2.1 Modular Arithmetic



Code 4.4: include/nn/nn.h

```c
#ifndef __NN_H__
#define __NN_H__

#include <crypto/words/words.h>
#include <crypto/nn/nn_config.h>

typedef struct {
    word_t val[BIT_LEN_WORDS(NN_MAX_BIT_LEN)];
    word_t magic;
    u8 wlen;
} nn;

typedef nn *nn_t;
typedef const nn *nn_src_t;

int nn_check_initialized(nn_src_t A);
int nn_is_initialized(nn_src_t A);
int nn_zero(nn_t A);
int nn_one(nn_t A);
int nn_set_word_value(nn_t A, word_t val);
void nn_uninit(nn_t A);
int nn_init(nn_t A, u16 len);
int nn_init_from_buf(nn_t A, const u8 *buf, u16 buflen);
int nn_cnd_swap(int cnd, nn_t in1, nn_t in2);
int nn_set_wlen(nn_t A, u8 new_wlen);
int nn_iszero(nn_src_t A, int *iszero);
int nn_isone(nn_src_t A, int *isone);
int nn_isodd(nn_src_t A, int *isodd);
int nn_cmp_word(nn_src_t in, word_t w, int *cmp);
int nn_cmp(nn_src_t A, nn_src_t B, int *cmp);
int nn_copy(nn_t dst_nn, nn_src_t src_nn);
int nn_normalize(nn_t in1);
int nn_export_to_buf(u8 *buf, u16 buflen, nn_src_t in_nn);
int nn_tabselect(nn_t out, u8 idx, nn_src_t *tab, u8 tabsize);

#endif /* __NN_H__ */
```

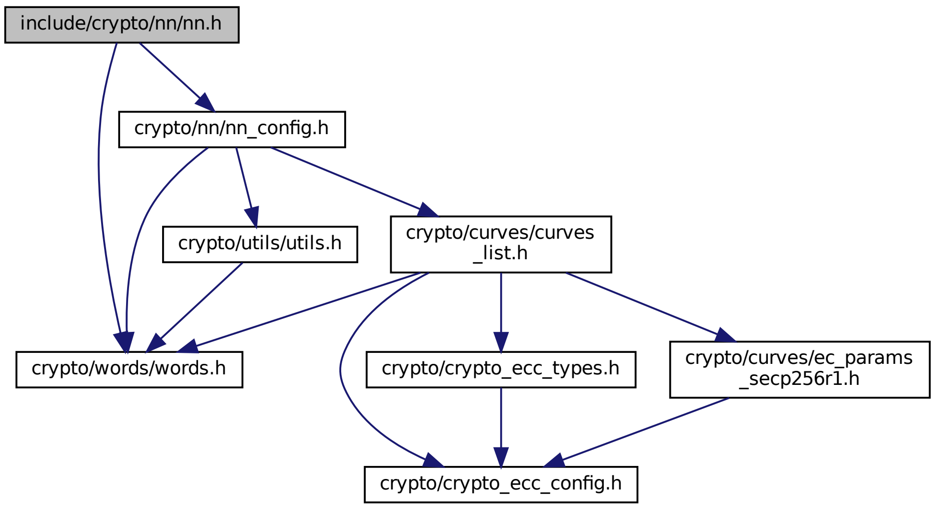
4.2.2 Finite Field



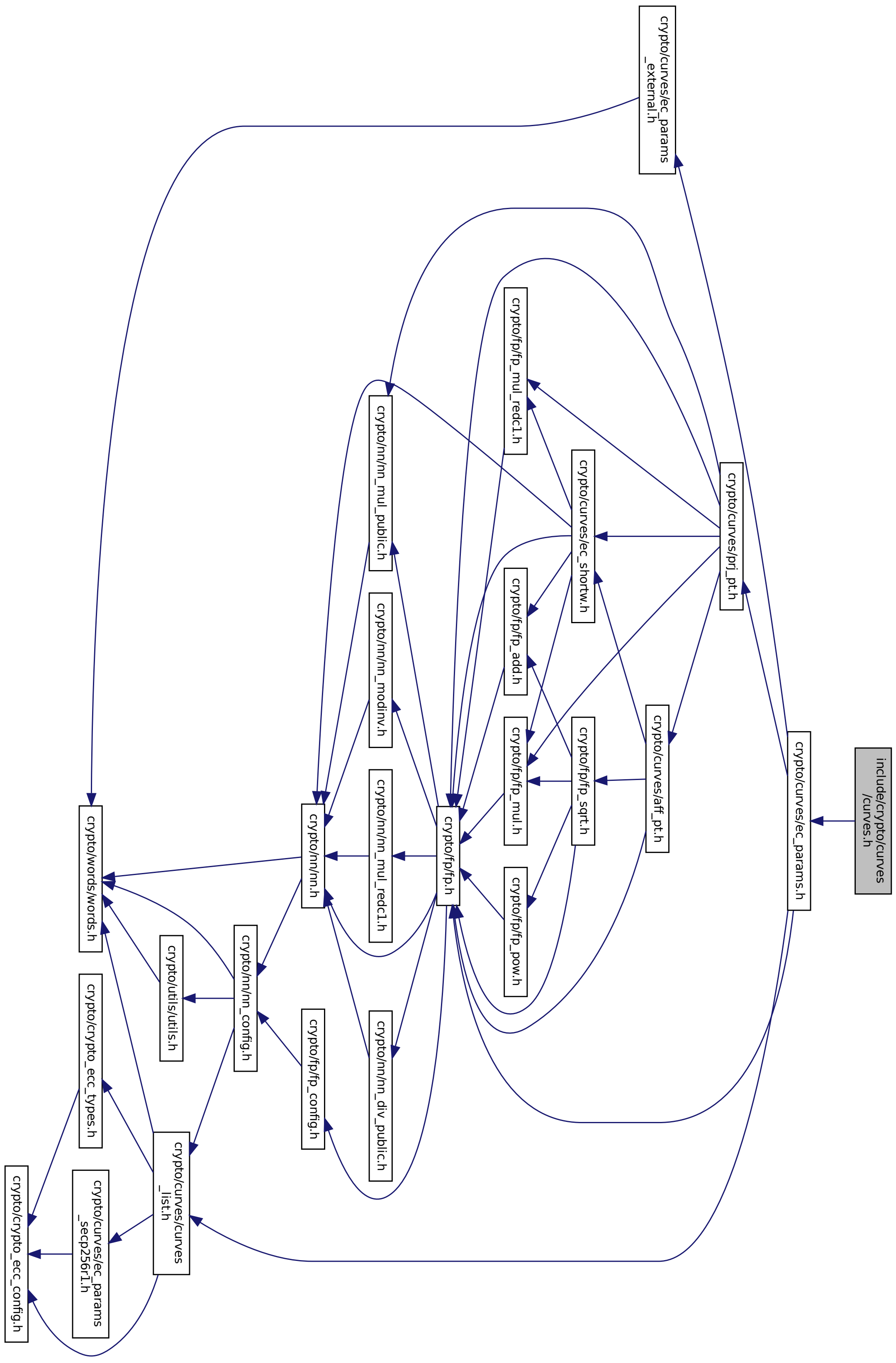
Code 4.5: include/fp/fp.h

```c
#ifndef __FP_H__
#define __FP_H__

#include <crypto/nn/nn.h>
#include <crypto/nn/nn_div_public.h>
#include <crypto/nn/nn_modinv.h>
#include <crypto/nn/nn_mul_public.h>
#include <crypto/nn/nn_mul_redc1.h>
#include <crypto/fp/fp_config.h>

typedef struct {
    /*
    * Value of p (extended by one word to handle overflows in Fp).
    * p_bitlen provides its length in bit.
    */
    nn p;
    bitcnt_t p_bitlen;

    word_t mpinv; /* -p^-1 mod 2^(bitsizeof(word_t)) */
    nn r; /* 2^bitsizeof(p) mod p */
    nn r_square; /* 2^(2*bitsizeof(p)) mod p */

    bitcnt_t p_shift;  /* clz(p) */
    nn p_normalized; /* p << p_shift */
    word_t p_reciprocal; /* floor(B^3/(DMSW(p_normalized) + 1)) - B */

    word_t magic;
} fp_ctx;

```

```c
typedef fp_ctx *fp_ctx_t;
typedef const fp_ctx *fp_ctx_src_t;

int fp_ctx_check_initialized(fp_ctx_src_t ctx);
int fp_ctx_init(fp_ctx_t ctx, nn_src_t p, bitcnt_t p_bitlen, nn_src_t r, nn_src_t
    ↪ r_square, word_t mpinv, bitcnt_t p_shift, nn_src_t p_normalized, word_t
    ↪ p_reciprocal);
int fp_ctx_init_from_p(fp_ctx_t ctx, nn_src_t p);

/* Then the definition of our Fp elements */
typedef struct {
    nn fp_val;
    fp_ctx_src_t ctx;
    word_t magic;
} fp;

typedef fp *fp_t;
typedef const fp *fp_src_t;

int fp_check_initialized(fp_src_t in);
int fp_init(fp_t A, fp_ctx_src_t fpctx);
int fp_init_from_buf(fp_t A, fp_ctx_src_t fpctx, const u8 *buf, u16 buflen);
void fp_uninit(fp_t A);
int fp_set_nn(fp_t out, nn_src_t in);
int fp_zero(fp_t out);
int fp_one(fp_t out);
int fp_set_word_value(fp_t out, word_t val);
int fp_cmp(fp_src_t in1, fp_src_t in2, int *cmp);
int fp_iszero(fp_src_t in, int *iszero);
int fp_copy(fp_t out, fp_src_t in);
int fp_tabselect(fp_t out, u8 idx, fp_src_t *tab, u8 tabsize);
int fp_eq_or_opp(fp_src_t in1, fp_src_t in2, int *eq_or_opp);
int fp_import_from_buf(fp_t out_fp, const u8 *buf, u16 buflen);
int fp_export_to_buf(u8 *buf, u16 buflen, fp_src_t in_fp);

#endif /* __FP_H__ */
```


include/crypto/nn/nn.h
crypto/nn/nn_config.h
crypto/utils/utils.h
crypto/curves/curves_list.h
crypto/words/words.h
crypto/crypto_ecc_types.h
crypto/curves/ec_params_secp256r1.h
crypto/crypto_ecc_config.h

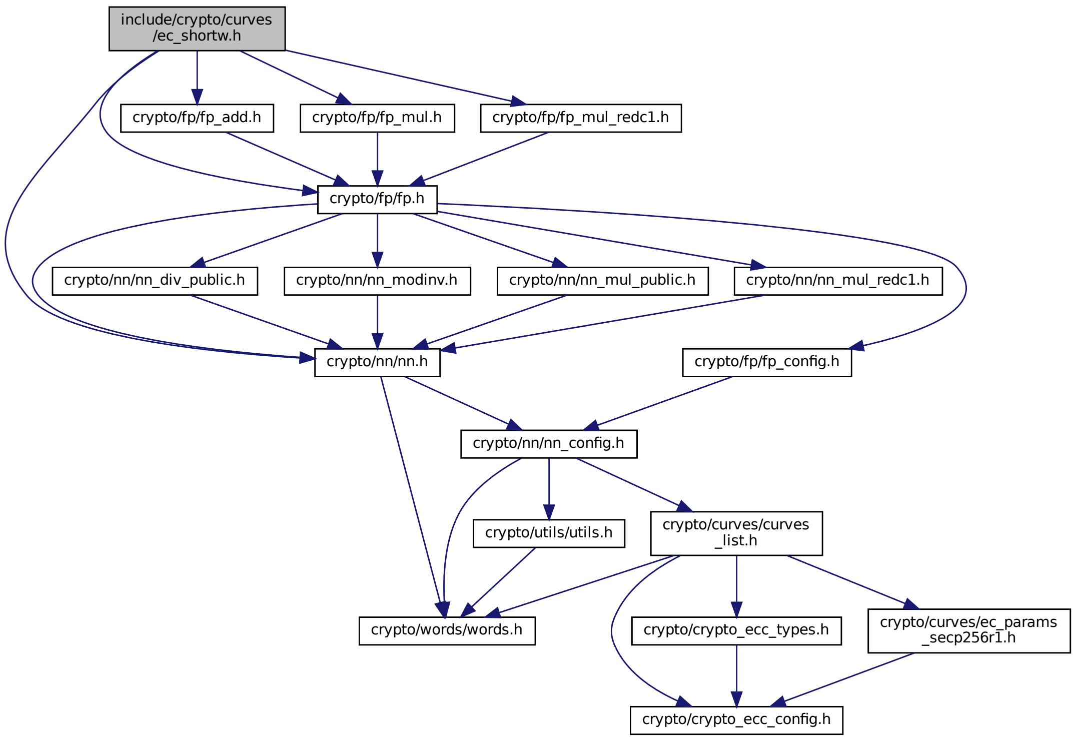
4.3 Elliptic Curve Library



4.3.1 Short Weierstrass Form

Code 4.6: include/curves/ec_shortw.h

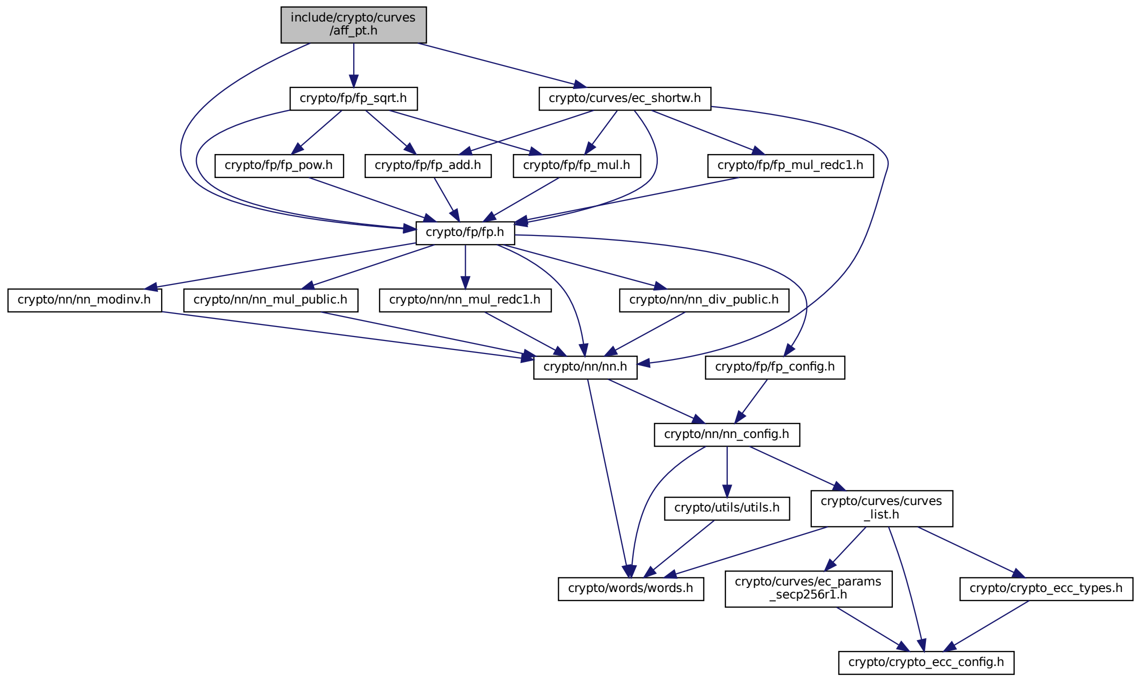
```c
#ifndef __EC_SHORTW_H__
#define __EC_SHORTW_H__

#include <crypto/nn/nn.h>
#include <crypto/fp/fp.h>
#include <crypto/fp/fp_add.h>
#include <crypto/fp/fp_mul.h>
#include <crypto/fp/fp_mul_redc1.h>

typedef struct {
    fp a; fp b; fp a_monty;
#ifndef NO_USE_COMPLETE_FORMULAS
    fp b3; fp b_monty; fp b3_monty;
#endif
    nn order; /* curve order */
    word_t magic;
} ec_shortw_crv;

typedef ec_shortw_crv *ec_shortw_crv_t;
typedef const ec_shortw_crv *ec_shortw_crv_src_t;

int ec_shortw_crv_check_initialized(ec_shortw_crv_src_t crv);
int ec_shortw_crv_init(ec_shortw_crv_t crv, fp_src_t a, fp_src_t b, nn_src_t order);
void ec_shortw_crv_uninit(ec_shortw_crv_t crv);

#endif /* __EC_SHORTW_H__ */
```



4.3.2 Points of Affine Coordinate

Code 4.7: include/curves/aff_pt.h

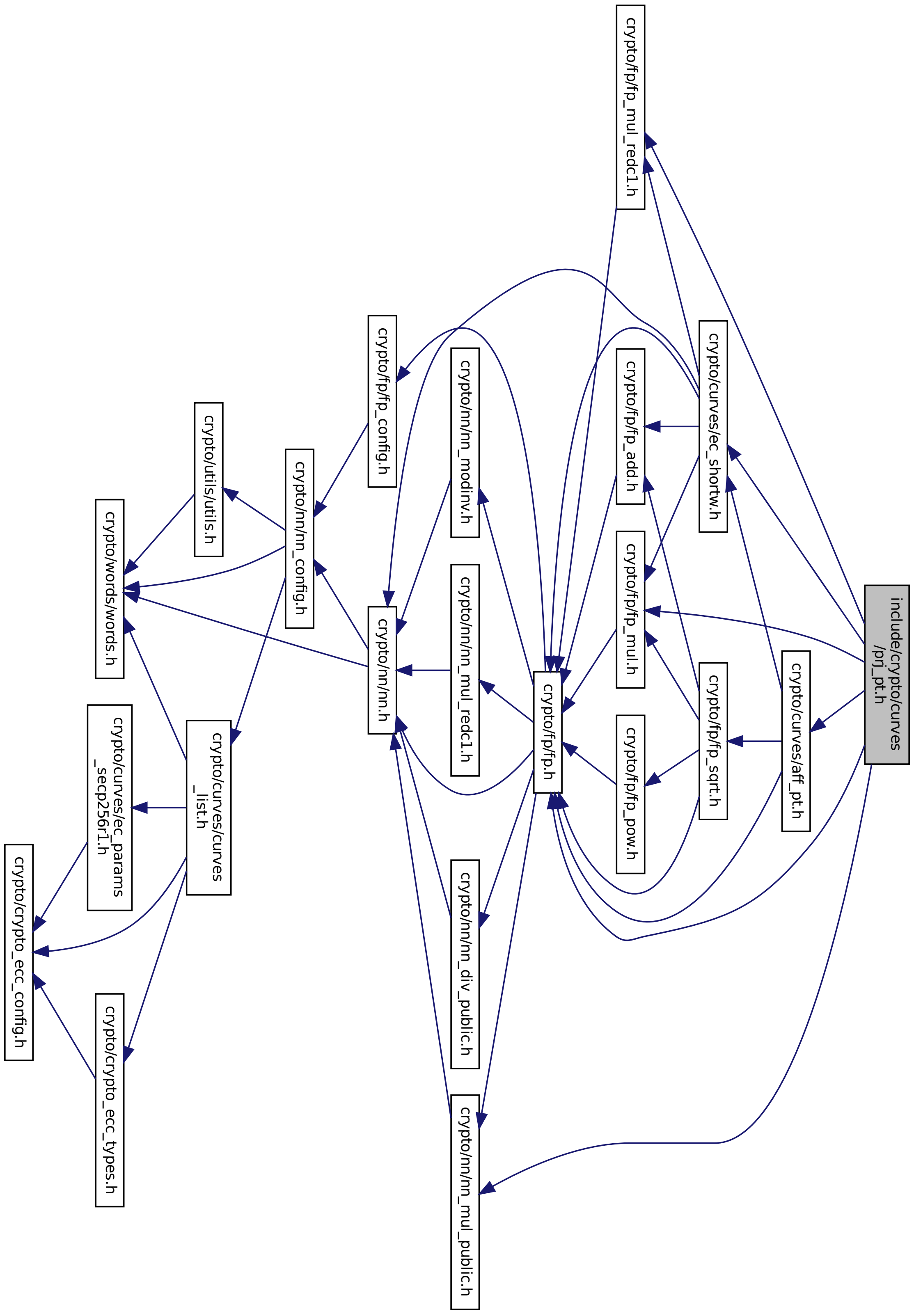
```c
#ifndef __AFF_PT_H__
#define __AFF_PT_H__
#include <crypto/fp/fp.h>
#include <crypto/fp/fp_sqrt.h>
#include <crypto/curves/ec_shortw.h>
typedef struct {
    fp x; fp y;
    ec_shortw_crv_src_t crv;
    word_t magic;
} aff_pt;
typedef aff_pt *aff_pt_t;
typedef const aff_pt_t aff_pt_src_t;
int aff_pt_check_initialized(aff_pt_src_t in);
int aff_pt_init(aff_pt_t in, ec_shortw_crv_src_t curve);
int aff_pt_init_from_coords(aff_pt_t in, ec_shortw_crv_src_t curve, fp_src_t xcoord, fp_src_t ycoord);
void aff_pt_uninit(aff_pt_t in);
int aff_pt_y_from_x(fp_t y1, fp_t y2, fp_src_t x, ec_shortw_crv_src_t curve);
int is_on_shortw_curve(fp_src_t x, fp_src_t y, ec_shortw_crv_src_t curve, int *on_curve);
int aff_pt_is_on_curve(aff_pt_src_t pt, int *on_curve);
int ec_shortw_aff_copy(aff_pt_t out, aff_pt_src_t in);
int ec_shortw_aff_cmp(aff_pt_src_t in1, aff_pt_src_t in2, int *cmp);
int ec_shortw_aff_eq_or_opp(aff_pt_src_t in1, aff_pt_src_t in2, int *eq_or_opp);
int aff_pt_import_from_buf(aff_pt_t pt, const u8 *pt_buf, u16 pt_buf_len, ec_shortw_crv_src_t crv);
int aff_pt_export_to_buf(aff_pt_src_t pt, u8 *pt_buf, u32 pt_buf_len);
#endif /* __AFF_PT_H__ */
```



4.3.3 Points of Projective Coordinate

Code 4.8: include/curves/prj_pt.h

```c
#ifndef __PRJ_PT_H__
#define __PRJ_PT_H__

#include <crypto/nn/nn_mul_public.h>
#include <crypto/fp/fp.h>
#include <crypto/fp/fp_mul.h>
#include <crypto/fp/fp_mul_redc1.h>
#include <crypto/curves/ec_shortw.h>
#include <crypto/curves/aff_pt.h>

typedef struct {
    fp X; fp Y; fp Z;
    ec_shortw_crv_src_t crv;
    word_t magic;
} prj_pt;

typedef prj_pt *prj_pt_t;
typedef const prj_pt *prj_pt_src_t;
typedef enum {
    PUBLIC_PT = 0,
    PRIVATE_PT = 1
} prj_pt_sensitivity;

int prj_pt_check_initialized(prj_pt_src_t in);
int prj_pt_init(prj_pt_t in, ec_shortw_crv_src_t curve);
int prj_pt_init_from_coords(prj_pt_t in, ec_shortw_crv_src_t curve, fp_src_t xcoord,
    ↪ fp_src_t ycoord, fp_src_t zcoord);
void prj_pt_uninit(prj_pt_t in);
int prj_pt_zero(prj_pt_t out);
int prj_pt_iszero(prj_pt_src_t in, int *iszero);
int prj_pt_is_on_curve(prj_pt_src_t in, int *on_curve);
int prj_pt_copy(prj_pt_t out, prj_pt_src_t in);
int prj_pt_to_aff(aff_pt_t out, prj_pt_src_t in);
int prj_pt_unique(prj_pt_t out, prj_pt_src_t in);
int ec_shortw_aff_to_prj(prj_pt_t out, aff_pt_src_t in);
int prj_pt_cmp(prj_pt_src_t in1, prj_pt_src_t in2, int *cmp);
int prj_pt_eq_or_opp(prj_pt_src_t in1, prj_pt_src_t in2, int *eq_or_opp);
int prj_pt_neg(prj_pt_t out, prj_pt_src_t in);
int prj_pt_add(prj_pt_t sum, prj_pt_src_t in1, prj_pt_src_t in2);
int prj_pt_dbl(prj_pt_t dbl, prj_pt_src_t in);
int prj_pt_mul(prj_pt_t out, nn_src_t m, prj_pt_src_t in);
int prj_pt_mul_blind(prj_pt_t out, nn_src_t m, prj_pt_src_t in);
/* XXX: WARNING: this function must only be used on public points! */
int _prj_pt_unprotected_mult(prj_pt_t out, nn_src_t cofactor, prj_pt_src_t public_in);
int check_prj_pt_order(prj_pt_src_t in_shortw, nn_src_t in_isorder,
    ↪ prj_pt_sensitivity s, int *check);
int prj_pt_import_from_buf(prj_pt_t pt, const u8 *pt_buf, u16 pt_buf_len,
    ↪ ec_shortw_crv_src_t crv);
int prj_pt_import_from_aff_buf(prj_pt_t pt, const u8 *pt_buf, u16 pt_buf_len,
    ↪ ec_shortw_crv_src_t crv);
int prj_pt_export_to_buf(prj_pt_src_t pt, u8 *pt_buf, u32 pt_buf_len);
int prj_pt_export_to_aff_buf(prj_pt_src_t pt, u8 *pt_buf, u32 pt_buf_len);

#endif /* __PRJ_PT_H__ */
```

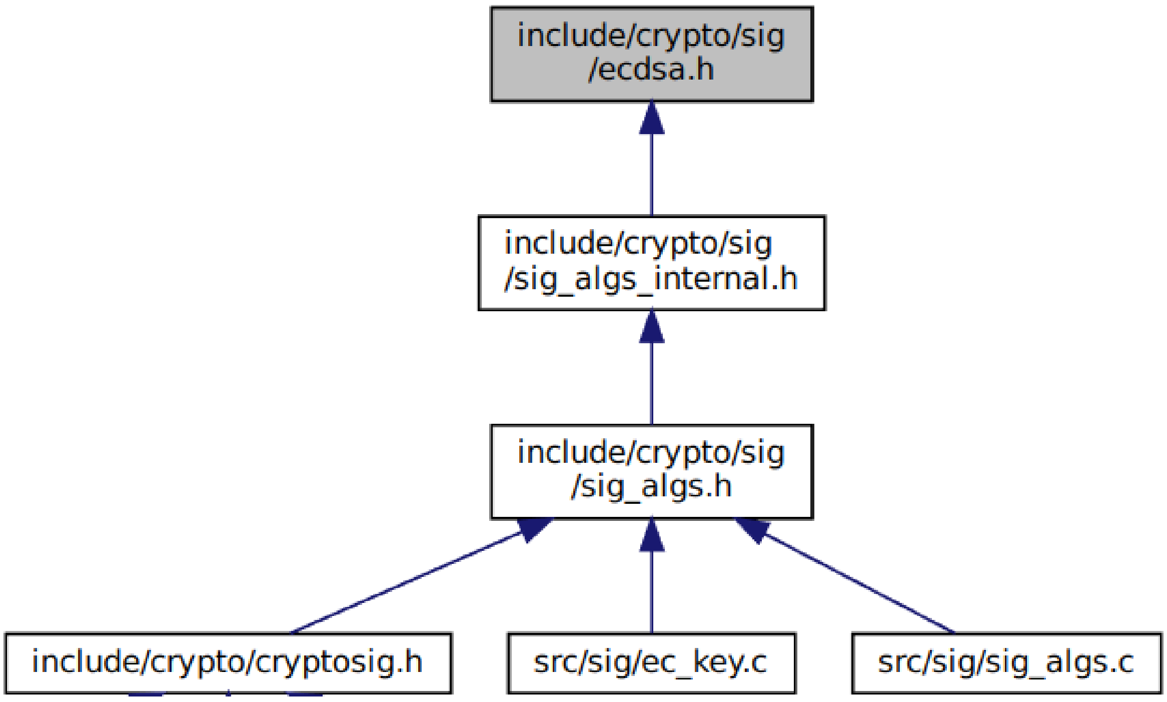

include/crypto/curves
/prj_pt.h
crypto/fp/fp_mul_redc1.h
crypto/curves/ec_shortw.h
crypto/curves/aff_pt.h
crypto/fp/fp_add.h
crypto/fp/fp_mul.h
crypto/fp/fp_sqrt.h
crypto/fp/fp_pow.h
crypto/fp/fp.h
crypto/nn/nn_modinv.h
crypto/nn/nn_mul_redc1.h
crypto/nn/nn_div_public.h
crypto/nn/nn_mul_public.h
crypto/fp/fp_config.h
crypto/nn/nn.h
crypto/nn/nn_config.h
crypto/utils/utils.h
crypto/curves/curves
_list.h
crypto/words/words.h
crypto/curves/ec_params
_secp256r1.h
crypto/crypto_ecc_config.h
crypto/crypto_ecc_types.h

4.4 Elliptic Curve Digital Signature Algorithm Library

Code 4.9: include/curves/prj_pt.h

```
#include <crypto/crypto_ecc_config.h>
#include <crypto/crypto_ecc_types.h>
#ifndef __ECDSA_COMMON_H__
#define __ECDSA_COMMON_H__

#include <crypto/words/words.h>
#include <crypto/sig/ec_key.h>
#include <crypto/hash/hash_algs.h>
#include <crypto/curves/curves.h>
#include <crypto/utils/utils.h>

#define ECDSA_R_LEN(q_bit_len) (BYTECEIL(q_bit_len))
#define ECDSA_S_LEN(q_bit_len) (BYTECEIL(q_bit_len))
#define ECDSA_SIGLEN(q_bit_len) (ECDSA_R_LEN(q_bit_len) + \
ECDSA_S_LEN(q_bit_len))
#define ECDSA_MAX_SIGLEN ECDSA_SIGLEN(CURVES_MAX_Q_BIT_LEN)

/*
* Compute max signature length for all the mechanisms enabled in the library (see
    ↪ lib_ecc_config.h).
* Having that done during
* preprocessing sadly requires some verbosity.
*/
#ifndef EC_MAX_SIGLEN
#define EC_MAX_SIGLEN 0
#endif
#if ((EC_MAX_SIGLEN) < (ECDSA_MAX_SIGLEN))
#undef EC_MAX_SIGLEN
#define EC_MAX_SIGLEN ECDSA_MAX_SIGLEN
#endif

typedef struct {
    hash_context h_ctx;
    word_t magic;
} ecdsa_sign_data;

struct ec_sign_context;

int __ecdsa_init_pub_key(ec_pub_key *out_pub, const ec_priv_key *in_priv, ec_alg_type
    ↪ key_type);

int __ecdsa_siglen(u16 p_bit_len, u16 q_bit_len, u8 hsize, u8 blocksize, u8 *siglen);
int __ecdsa_sign_init(struct ec_sign_context *ctx, ec_alg_type key_type);
int __ecdsa_sign_update(struct ec_sign_context *ctx,
const u8 *chunk, u32 chunklen, ec_alg_type key_type);
int __ecdsa_sign_finalize(struct ec_sign_context *ctx, u8 *sig, u8 siglen,
    ↪ ec_alg_type key_type);

typedef struct {
    nn r; nn s;
    hash_context h_ctx;
    word_t magic;
} ecdsa_verify_data;

```

```
struct ec_verify_context;

int __ecdsa_verify_init(struct ec_verify_context *ctx,
const u8 *sig, u8 siglen, ec_alg_type key_type);
int __ecdsa_verify_update(struct ec_verify_context *ctx,
const u8 *chunk, u32 chunklen, ec_alg_type key_type);
int __ecdsa_verify_finalize(struct ec_verify_context *ctx, ec_alg_type key_type);

int __ecdsa_public_key_from_sig(ec_pub_key *out_pub1, ec_pub_key *out_pub2, const
    ↪ ec_params *params, const u8 *sig, u8 siglen, const u8 *hash, u8 hsize,
    ↪ ec_alg_type key_type);

#endif /* __ECDSA_COMMON_H__ */
```


include/crypto/sig/ecdsa.h
include/crypto/sig/sig_algs_internal.h
include/crypto/sig/sig_algs.h
include/crypto/cryptosig.h
src/sig/ec_key.c
src/sig/sig_algs.c

4.5 Build with Makefile

This section describes the build system for generating (static) libraries, driven by a single GNU Makefile. It covers compiler settings, directory layout, source discovery, and all available targets.

Build without Makefile

```
mkdir -p obj/nn obj/fp obj/utils
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/nn/nn_add.o src/nn/nn_add.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/nn/nn.o src/nn/nn.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/nn/nn_div.o src/nn/nn_div.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/nn/nn_logical.o src/nn/
    ↪ nn_logical.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/nn/nn_modinv.o src/nn/
    ↪ nn_modinv.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/nn/nn_mod_pow.o src/nn/
    ↪ nn_mod_pow.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/nn/nn_mul.o src/nn/nn_mul.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/nn/nn_mul_redc1.o src/nn/
    ↪ nn_mul_redc1.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/nn/nn_rand.o src/nn/nn_rand.c

gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/fp/fp_add.o src/fp/fp_add.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/fp/fp.o src/fp/fp.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/fp/fp_mul.o src/fp/fp_mul.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/fp/fp_mul_redc1.o src/fp/
    ↪ fp_mul_redc1.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/fp/fp_pow.o src/fp/fp_pow.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/fp/fp_rand.o src/fp/fp_rand.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/fp/fp_sqrt.o src/fp/fp_sqrt.c

gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/utils/utils.o src/utils/utils
    ↪ .c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/utils/utils_rand.o src/utils/
    ↪ utils_rand.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/utils/print_nn.o src/utils/
    ↪ print_nn.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/utils/print_fp.o src/utils/
    ↪ print_fp.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/utils/print_buf.o src/utils/
    ↪ print_buf.c

mkdir -p obj/external_deps
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/external_deps/print.o src/
    ↪ external_deps/print.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/external_deps/rand.o src/
    ↪ external_deps/rand.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/external_deps/time.o src/
    ↪ external_deps/time.c

mkdir -p lib
ar rcs lib/libarith.a \
       obj/fp/fp_add.o obj/fp/fp.o obj/fp/fp_mul.o obj/fp/fp_mul_redc1.o obj/fp/fp_pow.o obj/fp/
           ↪ fp_rand.o obj/fp/fp_sqrt.o \
       obj/nn/nn_add.o obj/nn/nn.o obj/nn/nn_div.o obj/nn/nn_logical.o obj/nn/nn_modinv.o obj/nn
           ↪ /nn_mod_pow.o obj/nn/nn_mul.o obj/nn/nn_mul_redc1.o obj/nn/nn_rand.o \
       obj/utils/utils.o obj/utils/utils_rand.o obj/utils/print_nn.o obj/utils/print_fp.o obj/
           ↪ utils/print_buf.o \
       obj/external_deps/print.o obj/external_deps/rand.o obj/external_deps/time.o

mkdir -p obj/curves
```

```
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/curves/aff_pt.o src/curves/
    ↪ aff_pt.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/curves/curves.o src/curves/
    ↪ curves.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/curves/ec_params.o src/curves
    ↪ /ec_params.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/curves/ec_shortw.o src/curves
    ↪ /ec_shortw.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/curves/prj_pt.o src/curves/
    ↪ prj_pt.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/utils/print_curves.o src/
    ↪ utils/print_curves.c

ar rcs lib/libec.a \
       obj/fp/fp_add.o obj/fp/fp.o obj/fp/fp_mul.o obj/fp/fp_mul_redc1.o obj/fp/fp_pow.o obj/fp/
           ↪ fp_rand.o obj/fp/fp_sqrt.o \
       obj/nn/nn_add.o obj/nn/nn.o obj/nn/nn_div.o obj/nn/nn_logical.o obj/nn/nn_modinv.o obj/nn
           ↪ /nn_mod_pow.o obj/nn/nn_mul.o obj/nn/nn_mul_redc1.o obj/nn/nn_rand.o \
       obj/utils/utils.o obj/utils/utils_rand.o obj/utils/print_nn.o obj/utils/print_fp.o obj/
           ↪ utils/print_buf.o \
       obj/curves/aff_pt.o obj/curves/curves.o obj/curves/ec_params.o obj/curves/ec_shortw.o obj
           ↪ /curves/prj_pt.o \
       obj/utils/print_curves.o \
       obj/external_deps/print.o obj/external_deps/rand.o obj/external_deps/time.o

mkdir -p obj/hash obj/sig
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/hash/sha256.o src/hash/sha256
    ↪ .c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/hash/sha3-256.o src/hash/sha3
    ↪ -256.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/hash/sha3.o src/hash/sha3.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/hash/shake256.o src/hash/
    ↪ shake256.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/hash/shake.o src/hash/shake.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/hash/hash_algs.o src/hash/
    ↪ hash_algs.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/hash/hmac.o src/hash/hmac.c

gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/sig/ecdsa.o src/sig/ecdsa.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/sig/ecdsa_common.o src/sig/
    ↪ ecdsa_common.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/sig/sig_algs.o src/sig/
    ↪ sig_algs.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/sig/ec_key.o src/sig/ec_key.c
gcc -MMD -MP -c -O2 -std=c99 -Wall -Wextra -Iinclude -fPIC -o obj/utils/print_keys.o src/utils/
    ↪ print_keys.c

ar rcs lib/libsign.a \
       obj/fp/fp_add.o obj/fp/fp.o obj/fp/fp_mul.o obj/fp/fp_mul_redc1.o obj/fp/fp_pow.o obj/fp/
           ↪ fp_rand.o obj/fp/fp_sqrt.o \
       obj/nn/nn_add.o obj/nn/nn.o obj/nn/nn_div.o obj/nn/nn_logical.o obj/nn/nn_modinv.o obj/nn
           ↪ /nn_mod_pow.o obj/nn/nn_mul.o obj/nn/nn_mul_redc1.o obj/nn/nn_rand.o \
       obj/utils/utils.o obj/utils/utils_rand.o obj/utils/print_nn.o obj/utils/print_fp.o obj/
           ↪ utils/print_buf.o \
       obj/curves/aff_pt.o obj/curves/curves.o obj/curves/ec_params.o obj/curves/ec_shortw.o obj
           ↪ /curves/prj_pt.o \
       obj/utils/print_curves.o \
       obj/hash/sha256.o obj/hash/sha3-256.o obj/hash/sha3.o obj/hash/shake256.o obj/hash/shake.
           ↪ o obj/hash/hash_algs.o obj/hash/hmac.o \
       obj/sig/ecdsa.o obj/sig/ecdsa_common.o obj/sig/sig_algs.o obj/sig/ec_key.o \
       obj/utils/print_keys.o \
       obj/external_deps/print.o obj/external_deps/rand.o obj/external_deps/time.o
```

Build with Makefile

[gcc (GNU Compiler Collection)]

- gcc: compiles and links source code.

[ar rcs (GNU Archiver)]

- ar rcs: creates or updates static library archives.
  - r: "replace" – add or replace files in the archive.
  - c: "create" – silently create the archive if it doesn't exist.
  - s: "index" – write an index (symbol table) for faster linking.

  Creates and maintains static library archives (.a) from object files.

```
#-----------------------------------------------------------------------------
# Makefile: separate obj/, lib/, bin/; builds libarith.a, libec.a, libsign.a
#-----------------------------------------------------------------------------

# Directories
OBJ_DIR := obj
LIB_DIR := lib
BIN_DIR := bin
INCLUDE := include
SRC_DIR := src

# Compiler & tools
CC := gcc
AR := ar rcs

# Flags
CFLAGS := -MMD -MP -O3 -std=c99 -Wall -Wextra -I$(INCLUDE)
PICFLAGS := -fPIC

# Module source files
FP_SRCS := $(wildcard $(SRC_DIR)/fp/fp_*.c)
NN_SRCS := $(wildcard $(SRC_DIR)/nn/nn_*.c) $(SRC_DIR)/nn/nn.c
UTILS_ARITH := $(SRC_DIR)/utils/utils.c $(SRC_DIR)/utils/utils_rand.c
UTILS_PRINT := $(wildcard $(SRC_DIR)/utils/print_*.c)
CURVES_SRCS := $(wildcard $(SRC_DIR)/curves/*.c)
HASH_SRCS := $(wildcard $(SRC_DIR)/hash/*.c)
SIG_SRCS := $(SRC_DIR)/sig/ecdsa.c $(SRC_DIR)/sig/ecdsa_common.c $(SRC_DIR)/sig/sig_algs.c $(
    ↪ SRC_DIR)/sig/ec_key.c

# Generate object lists by mirroring src/ to obj/
define to_obj
       $(patsubst $(SRC_DIR)/%, $(OBJ_DIR)/%, $(1:.c=.o))
endef

FP_OBJS := $(call to_obj,$(FP_SRCS))
NN_OBJS := $(call to_obj,$(NN_SRCS))
UTILS_ARITH_OBJS := $(call to_obj,$(UTILS_ARITH))
UTILS_PRINT_OBJS := $(call to_obj,$(UTILS_PRINT))
CURVES_OBJS := $(call to_obj,$(CURVES_SRCS))
HASH_OBJS := $(call to_obj,$(HASH_SRCS))
SIG_OBJS := $(call to_obj,$(SIG_SRCS))

# Library object groups
LIBARITH_OBJS := $(FP_OBJS) $(NN_OBJS) $(UTILS_ARITH_OBJS) $(UTILS_PRINT_OBJS)
LIBEC_OBJS := $(LIBARITH_OBJS) $(CURVES_OBJS)
```

```
LIBSIGN_OBJS := $(LIBEC_OBJS) $(HASH_OBJS) $(SIG_OBJS)

# All objects and deps
ALL_OBJS := $(sort $(LIBSIGN_OBJS))
DEPS := $(ALL_OBJS:.o=.d)

# Phony targets
.PHONY: all clean rebuild
all: $(LIB_DIR)/libarith.a $(LIB_DIR)/libec.a $(LIB_DIR)/libsign.a

clean:
	rm -rf $(OBJ_DIR) $(LIB_DIR) $(BIN_DIR)
	rm -f *~

# Compile rule: .c => .o (+ deps)
$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c
	@mkdir -p $(dir $@)
	$(CC) $(CFLAGS) $(PICFLAGS) -MMD -MP -c $< -o $@

# Static libraries
$(LIB_DIR)/libarith.a: $(LIBARITH_OBJS)
	@mkdir -p $(LIB_DIR)
	$(AR) $@ $^

$(LIB_DIR)/libec.a: $(LIBEC_OBJS)
	@mkdir -p $(LIB_DIR)
	$(AR) $@ $^

$(LIB_DIR)/libsign.a: $(LIBSIGN_OBJS)
	@mkdir -p $(LIB_DIR)
	$(AR) $@ $^

rebuild:
	$(MAKE) clean
	$(MAKE) -j8

# Include dependency files
-include $(DEPS)
```

- `-MMD`: generates dependency files (`.d`) for user headers
- `-MP`: adds phony targets for missing headers.
- `-O3`: enables high-level optimizations.
- `-std=c99`: enforces ISO C99 compliance.
- `-Wall`: enables a broad set of common warning checks to catch potential issues early.
- `-Wextra`: enables additional, more pedantic warnings beyond `-Wall`.
- `-Iinclude`: adds the `include/` directory to the header search path.
- `-fPIC`: generates position-independent code for use in shared libraries, allowing flexible load addresses.

```
@:~$ cd lib
@:~$ ls -lh
total 552K
-rw-rw-r-- 1 hacker-code-j hacker-code-j 123K Jun 11 06:58 libarith.a
-rw-rw-r-- 1 hacker-code-j hacker-code-j 176K Jun 11 06:58 libec.a
-rw-rw-r-- 1 hacker-code-j hacker-code-j 251K Jun 11 06:58 libsign.a
```

4.6 Sample Code

Code 4.10: sample_ecdsa.c

```c
/*
 * sample_ecdsa.c
 *
 * A minimal example of using libecc's ECDSA on NIST P-256 to sign and verify
 * a simple message string.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdint.h>

#include <crypto/cryptosig.h> // For ECDSA

// Hex-encode buffer
static char *to_hex(const uint8_t *buf, size_t len) {
    static const char hex[] = "0123456789ABCDEF";
    char *out = malloc(len*2 + 1);
    for (size_t i = 0; i < len; i++) {
        out[2*i] = hex[buf[i] >> 4];
        out[2*i+1] = hex[buf[i] & 0xF];
    }
    out[len*2] = '\0';
    return out;
}

int main(void) {
    // 1. Load P-256 parameters
    ec_params params;
    import_params(&params, &secp256r1_str_params);

    // 2. Generate key pair
    ec_key_pair kp;
    if (ec_key_pair_gen(&kp, &params, ECDSA) != 0) {
        fprintf(stderr, "Key generation failed\n");
        return 1;
    }

    // 3. Prepare message and hash it
    const char *msg = "Hello, ECDSA!";
    const hash_mapping *hm;
    if (get_hash_by_type(SHA256, &hm) != 0) {
        fprintf(stderr, "SHA-256 not available\n");
        return 1;
    }
    hash_context hctx;
    hm->hfunc_init(&hctx);
    hm->hfunc_update(&hctx, (const uint8_t*)msg, (u32)strlen(msg));
    uint8_t digest[64];
    u8 dlen = hm->digest_size;
    hm->hfunc_finalize(&hctx, digest);

    // 4. Sign the digest
    u8 siglen;
    ec_get_sig_len(&params, ECDSA, SHA256, &siglen);
    uint8_t *sigbin = malloc(siglen);
    if (ec_sign(sigbin, siglen,
    &kp, digest, dlen,
    ECDSA, SHA256,
    NULL, 0) != 0)
    {
        fprintf(stderr, "Signing failed\n");
        free(sigbin);
        return 1;
    }

    char *hexsig = to_hex(sigbin, siglen);
    printf("Signature: %s\n", hexsig);

    // 5. Derive public key and verify
    ec_pub_key pub;
    if (init_pubkey_from_privkey(&pub, &kp.priv_key) != 0) {
        fprintf(stderr, "Public key derivation failed\n");
        free(sigbin);
        free(hexsig);
        return 1;
    }

    int ok = ec_verify(sigbin, siglen,
    &pub, digest, dlen,
    ECDSA, SHA256,
    NULL, 0);
    printf("Verification: %s\n", (ok == 0) ? "SUCCESS" : "FAILURE");

    // Cleanup
    free(sigbin);
    free(hexsig);
    return (ok == 0) ? 0 : 1;
}
```

```
@:~$ gcc -std=c99 -D_POSIX_C_SOURCE=200809L -O2 -Iinclude \
      sample_ecdsa.c -o sample_ecdsa \
      lib/libarith.a lib/libec.a lib/libsign.a
@:~$ ./sample_ecdsa
Signature: 0934EE14D27FB767668EE5EA1E850165B47C0624D34B74D53E77846FE868A042A036D3AA4E2C67EB1
F8EE2F6948C68378CF88E2E04CB48EB63CEA78AE4D14FF9
Verification: SUCCESS
```

Chapter 5

Application 1: The File Checker Tool

The `filecheck_ecc` utility is a command-line application designed to provide cryptographic integrity and authenticity guarantees for file-based data. It serves as a foundational tool for securing data at rest, enabling users to generate key pairs, sign files of any size, and verify existing signatures.

5.1 Implementation of File Checker

```c
/*
* filecheck_ecc.c
*
* Sign and verify large files using ECDSA on SECP256R1.
*
* Usage:
* filecheck_ecc generate <key-base>
* filecheck_ecc sign <key-base>_priv.bin <in-file> <sig-file>
* filecheck_ecc verify <key-base>_pub.bin <in-file> <sig-file>
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <crypto/cryptosig.h>

#define CHUNK_SIZE 4096

// #define SHA_ALG SHA256
#define SHA_ALG SHA3_256

/* Hex-encode a buffer */
static char *to_hex(const uint8_t *buf, size_t len) {
    static const char hex[] = "0123456789ABCDEF";
    char *out = (char *)malloc(len*2 + 1);
    for (size_t i = 0; i < len; i++) {
        out[2*i] = hex[buf[i] >> 4];
        out[2*i+1] = hex[buf[i] & 0xF];
    }
    out[len*2] = '\0';
    return out;
}

```

```c
/* Trim CR/LF from end of string */
static void trim_newline(char *s) {
    size_t n = strlen(s);
    while (n && (s[n-1]=='\n' || s[n-1]=='\r')) s[--n] = '\0';
}

/* Stream-hash 'path' via SHA-256 */
static int stream_hash(const char *path, uint8_t *digest, uint8_t *dlen) {
    const hash_mapping *hm;
    if (get_hash_by_type(SHA_ALG, &hm) != 0) return -1;
    hash_context ctx;
    if (hm->hfunc_init(&ctx) != 0) return -1;

    FILE *f = fopen(path, "rb");
    if (!f) return -1;
    uint8_t buf[CHUNK_SIZE];
    size_t n;
    while ((n = fread(buf,1,CHUNK_SIZE,f)) > 0) {
        hm->hfunc_update(&ctx, buf, (u32)n);
    }
    fclose(f);

    *dlen = hm->digest_size;
    return hm->hfunc_finalize(&ctx, digest);
}

/* generate "<base>_priv.bin", "<base>_pub.bin" */
static int do_generate(const char *base) {
    ec_params params;
    import_params(&params, &secp256r1_str_params);

    ec_key_pair kp;
    if (ec_key_pair_gen(&kp, &params, ECDSA) != 0) {
        fprintf(stderr, "Error: key generation failed\n"); return -1;
    }

    /* Export private key */
    u8 priv_sz = EC_STRUCTURED_PRIV_KEY_EXPORT_SIZE(&kp.priv_key);
    u8 *priv_buf = (u8 *)malloc(priv_sz);
    ec_structured_priv_key_export_to_buf(&kp.priv_key, priv_buf, priv_sz);

    char fn[256];
    snprintf(fn, sizeof(fn), "%s_priv.bin", base);
    FILE *f = fopen(fn, "wb");
    fwrite(priv_buf,1,priv_sz,f);
    fclose(f);
    free(priv_buf);

    /* Export public key */
    u8 pub_sz = EC_STRUCTURED_PUB_KEY_EXPORT_SIZE(&kp.pub_key);
    u8 *pub_buf = (u8 *)malloc(pub_sz);
    ec_structured_pub_key_export_to_buf(&kp.pub_key, pub_buf, pub_sz);

    snprintf(fn, sizeof(fn), "%s_pub.bin", base);
    f = fopen(fn, "wb");
    fwrite(pub_buf,1,pub_sz,f);
    fclose(f);
    free(pub_buf);

    printf("Key pair generated:\n"
    " Secret key: %s_priv.bin\n"
    " Public key: %s_pub.bin\n", base, base);
    return 0;
}

/* sign infile using privfile, write hex-DER to sigfile */
static int do_sign(const char *privfile, const char *infile, const char *sigfile) {
    /* load priv blob */
    FILE *f = fopen(privfile, "rb");
    if (!f) { perror(privfile); return -1; }
    fseek(f,0,SEEK_END);
    size_t sz = ftell(f);
    fseek(f,0,SEEK_SET);
    u8 *buf = (u8 *)malloc(sz);
    fread(buf,1,sz,f);
    fclose(f);

    ec_params params;
    import_params(&params, &secp256r1_str_params);
    ec_key_pair kp;
    // 1) import *only* the private key
    if (ec_structured_priv_key_import_from_buf(
        &kp.priv_key, &params, buf, (u8)sz, ECDSA) != 0)
    {
        fprintf(stderr, "Error: invalid private key\n");
        free(buf); return -1;
    }
    // 2) derive the public key from it
    if (init_pubkey_from_privkey(&kp.pub_key, &kp.priv_key) != 0) {
        fprintf(stderr, "Error: failed to derive public key\n"); return -1;
    }
    free(buf);

    /* hash file */
    uint8_t digest[64]; u8 dlen;
    if (stream_hash(infile, digest, &dlen) != 0) {
        fprintf(stderr, "Error: hashing failed\n"); return -1;
    }

    /* sign */
    u8 siglen;
    ec_get_sig_len(&params, ECDSA, SHA_ALG, &siglen);
    u8 *sigbin = (u8 *)malloc(siglen);
    if (ec_sign(sigbin, siglen, &kp, digest, dlen, ECDSA, SHA_ALG, NULL, 0) != 0) {
        fprintf(stderr, "Error: signing failed\n");
        free(sigbin); return -1;
    }

    /* hex + write */
    char *hex = to_hex(sigbin, siglen);
    f = fopen(sigfile, "w");
    fprintf(f, "%s\n", hex);
    fclose(f);

    free(sigbin);
    free(hex);
```

```c
    printf("Signed '%s' => '%s'\n", infile, sigfile);
    return 0;
}

/* verify infile against hex-DER sig in sigfile using pubfile */
static int do_verify(const char *pubfile, const char *infile, const char *sigfile) {
    /* load pub blob */
    FILE *f = fopen(pubfile, "rb");
    if (!f) { perror(pubfile); return -1; }
    fseek(f,0,SEEK_END);
    size_t sz = ftell(f);
    fseek(f,0,SEEK_SET);
    u8 *buf = (u8 *)malloc(sz);
    fread(buf,1,sz,f);
    fclose(f);

    ec_params params;
    import_params(&params, &secp256r1_str_params);
    ec_pub_key pub;
    // if (ec_structured_pub_key_import_from_buf(
    // &pub, &params, buf, (u8)sz, ECDSA) != 0)
    if (ec_structured_pub_key_import_from_buf(
    &pub, &params, buf, (u8)sz, ECDSA) != 0)
    {
        fprintf(stderr, "Error: invalid public key\n");
        free(buf); return -1;
    }
    free(buf);

    /* hash file */
    uint8_t digest[64]; u8 dlen;
    if (stream_hash(infile, digest, &dlen) != 0) {
        fprintf(stderr, "Error: hashing failed\n"); return -1;
    }

    /* read + decode hex sig */
    char *line = NULL; size_t cap=0;
    f = fopen(sigfile, "r");
    getline(&line, &cap, f);
    fclose(f);
    trim_newline(line);
    size_t sl = strlen(line)/2;
    u8 *sigbin = (u8 *)malloc(sl);
    for (size_t i = 0; i < sl; i++)
    sscanf(line + 2*i, "%2hhx", &sigbin[i]);
    free(line);

    /* verify */
    int rc = ec_verify(sigbin, (u8)sl,
    &pub, digest, dlen,
    ECDSA, SHA_ALG, NULL, 0);
    free(sigbin);
    if (rc == 0) {
        printf("VALID signature for '%s'\n", infile); return 0;
    } else {
        printf("INVALID signature for '%s'\n", infile); return 1;
    }
}
```

```c

int main(int argc, char **argv) {
    if (argc < 2) goto usage;
    if (!strcmp(argv[1], "generate") && argc == 3)
    return do_generate(argv[2]) ? 1 : 0;
    if (!strcmp(argv[1], "sign") && argc == 5)
    return do_sign(argv[2], argv[3], argv[4]) ? 1 : 0;
    if (!strcmp(argv[1], "verify") && argc == 5)
    return do_verify(argv[2], argv[3], argv[4]) ? 1 : 0;

    usage:
    fprintf(stderr,
    "Usage:\n"
    " %s generate <keybase>\n"
    " %s sign <priv> <infile> <sigfile>\n"
    " %s verify <pub> <infile> <sigfile>\n",
    argv[0], argv[0], argv[0]);
    return 1;
}
```

```
mkdir -p obj bin
gcc -std=c99 -Wall -Wextra -O2 -Iinclude -c src/filecheck_ecc.c -o obj/filecheck_ecc.o
gcc -std=c99 -Wall -Wextra -O2 -Iinclude -o bin/filecheck_ecc obj/filecheck_ecc.o \
      lib/libarith.a lib/libec.a lib/libsign.a
```

5.2 Example Scenario: File Checking

5.2.1 Key Pair Generation (generate)

This mode creates a new ECDSA key pair (a private key and its corresponding public key) using the SECP256R1 curve. The keys are then saved to disk in a simple, portable format.

```
@:~$ ./bin/filecheck_ecc generate mykey
Key pair generated:
      Secret key: mykey_priv.bin
      Public key: mykey_pub.bin
@:~$ hexdump -x mykey_priv.bin
0000000 0101 1204 98e9 ffe0 4d0f 82f1 3749 0f99
0000010 205e f772 0bbd 74db dadf 9c09 2efc 1947
0000020 b726 00ca
0000023
@:~$ hexdump -x mykey_pub.bin
0000000 0100 b604 246a 3768 298e 54dd 5ada cac8
0000010 54d3 b85c 9908 b657 f351 eda0 4a07 59b9
0000020 446a 2856 2007 96f5 6fe9 9335 3e73 d9b2
0000030 4eba b504 41e1 3c83 33d6 afdc 0652 5a1f
0000040 552b ccbd 56c2 47a9 7f83 3356 3656 7cd1
0000050 f4e1 8cde 72c3 675b ceda 1401 2dc0 5329
0000060 92c3 0009
0000063
```

5.2.2 File Signing (sign)

In this mode, the tool uses a specified private key to generate a digital signature for a given input file. The process involves first computing the SHA-256 hash of the input file and then signing that hash with the private key. The resulting signature is written to a separate output file.

```python
"""
generate_file.py

Generate a large random binary file of a specified size.
Usage:
python3 random_file.py <output_file> <size> [--chunk <chunk_size>]
e.g., python3 random_file.py largefile.bin 100M --chunk 1M
e.g., python3 random_file.py largefile.bin 4G --chunk 8M
"""

import os
import argparse

def parse_size(size_str):
    """
    Parse a human-friendly size string (e.g. '10M', '2G', '512K') into an integer number of bytes.
    """
    size_str = size_str.strip().upper()
    if size_str.endswith('G'):
        return int(float(size_str[:-1]) * 1024**3)
    if size_str.endswith('M'):
        return int(float(size_str[:-1]) * 1024**2)
    if size_str.endswith('K'):
        return int(float(size_str[:-1]) * 1024)
    return int(size_str)

def generate_random_file(path, total_size, chunk_size=1_048_576):
    """
    Write `total_size` random bytes to `path`, in chunks of `chunk_size` bytes.
    """
    written = 0
    with open(path, 'wb') as f:
        while written < total_size:
            to_write = min(chunk_size, total_size - written)
            f.write(os.urandom(to_write))
            written += to_write
    print(f"Done: wrote {written} bytes to {path!r}")

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Generate a large random binary file.")
    parser.add_argument('output', help="Output filename (e.g. largefile.bin)")
    parser.add_argument(
        'size',
        help="Total size (e.g. 100M for 100 megabytes, 2G for 2 gigabytes)"
    )
    parser.add_argument(
        '--chunk',
        default='1M',
        help="Chunk size for each write (default: 1M)"
    )
    args = parser.parse_args()

    total_bytes = parse_size(args.size)
    chunk_bytes = parse_size(args.chunk)
    generate_random_file(args.output, total_bytes, chunk_bytes)
```

```
@:~$ python3 src/generate_file.py largefile.bin 1G --chunk 32M
Done: wrote 1073741824 bytes to 'largefile.bin'

@:~$ ./bin/filecheck_ecc sign mykey_priv.bin largefile.bin largefile.sig
Signed 'largefile.bin' => 'largefile.sig'
```

5.2.3 Signature Verification (`verify`)

This mode performs the inverse operation of signing. It takes a public key, an input file, and a signature file as arguments. It re-computes the SHA-256 hash of the input file and uses the provided public key to verify that the signature corresponds to the computed hash. It then reports to the user whether the signature is VALID or INVALID.

```
@:~$ ./bin/filecheck_ecc verify mykey_pub.bin largefile.bin largefile.sig
VALID signature for 'largefile.bin'
```

Chapter 6

Application 2: The Logging Suite

While `filecheck_ecc` is designed for securing static, file-based data, modern big data systems rely heavily on processing continuous streams of data, such as application logs, metrics, and events. To address this domain, the Logging Suite was developed. This suite consists of two specialized, interoperable utilities designed to provide cryptographic integrity for streaming data: `logtool_ecc`, for creating individual signed log entries, and `logaggregator_ecc`, for signing a high-throughput stream of log data in real-time.

6.1 Implementation of Logging Suite

logtool_ecc.c

```c
/*
* src/logtool_ecc.c
*
* A Linux command-line tool for signing and verifying structured log entries
* using libecc's real ECDSA on P-256 (SECP256R1).
*
* Each log entry is a line with four '|'-separated fields:
* timestamp|level|message|hex_signature
*
* Usage:
* logtool_ecc generate <keybase>
* logtool_ecc write <privkey_bin> <logfile> <level> <message...>
* logtool_ecc verify <pubkey_bin> <logfile>
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <stdint.h>

#include <crypto/cryptosig.h>

/* Hex-encode a buffer */
static char *to_hex(const uint8_t *buf, size_t len) {
    static const char hex[] = "0123456789ABCDEF";
    char *out = malloc(len*2 + 1);
    for (size_t i = 0; i < len; i++) {
        out[2*i] = hex[buf[i] >> 4];
        out[2*i+1] = hex[buf[i] & 0xF];
    }
    out[len*2] = '\0';
    return out;
}

/* Trim CR/LF from end of string */
static void trim_nl(char *s) {
    size_t n = strlen(s);
    while (n && (s[n-1]=='\n' || s[n-1]=='\r')) s[--n] = '\0';
}

/* Write current timestamp "YYYY-MM-DD HH:MM:SS" into buf */
static void current_timestamp(char *buf, size_t bufsz) {
    time_t t = time(NULL);
    struct tm tm;
    localtime_r(&t, &tm);
    strftime(buf, bufsz, "%Y-%m-%d %H:%M:%S", &tm);
}

/* Print usage */
static void usage(const char *prog) {
    fprintf(stderr,
    "Usage:\n"
    " %s generate <keybase>\n"
    " %s write <privkey> <logfile> <level> <message...>\n"
    " %s verify <pubkey> <logfile>\n",
    prog, prog, prog);
}

/* Generate and save a keypair */
static int do_generate(const char *base) {
    // const ec_params *params = ec_maps[SECP256R1].params;
    ec_params params;
    import_params(&params, &secp256r1_str_params);

    ec_key_pair kp;
    if (ec_key_pair_gen(&kp, &params, ECDSA) != 0) {
        fprintf(stderr, "Error: key generation failed\n"); return -1;
    }

    /* Export private key */
    u8 priv_sz = EC_STRUCTURED_PRIV_KEY_EXPORT_SIZE(&kp.priv_key);
    u8 *priv_buf = malloc(priv_sz);
    ec_structured_priv_key_export_to_buf(&kp.priv_key, priv_buf, priv_sz);

    char fn[256];
    snprintf(fn, sizeof(fn), "%s_priv.bin", base);
    FILE *f = fopen(fn, "wb");
    if (!f) { perror(fn); return -1; }
    if (fwrite(priv_buf,1,priv_sz,f) != priv_sz) {
        fprintf(stderr, "Error: writing %s\n", fn);
        fclose(f); return -1;
    }
    fclose(f);
    free(priv_buf);

    /* Export public key */
    u8 pub_sz = EC_STRUCTURED_PUB_KEY_EXPORT_SIZE(&kp.pub_key);
    u8 *pub_buf = malloc(pub_sz);
    ec_structured_pub_key_export_to_buf(&kp.pub_key, pub_buf, pub_sz);

    snprintf(fn, sizeof(fn), "%s_pub.bin", base);
    f = fopen(fn, "wb");
    if (!f) { perror(fn); return -1; }
    if (fwrite(pub_buf,1,pub_sz,f) != pub_sz) {
        fprintf(stderr, "Error: writing %s\n", fn);
        fclose(f); return -1;
    }
    fclose(f);
    free(pub_buf);

    printf("Wrote %s_priv.bin and %s_pub.bin\n", base, base);
    return 0;
}

/* Write a signed log entry */
static int do_write(const char *privfile,
const char *logfile,
const char *level,
const char *message)
{
    /* Load private key blob */
    FILE *f = fopen(privfile, "rb");
    if (!f) { perror(privfile); return -1; }
    fseek(f,0,SEEK_END);
    size_t sz = ftell(f);
    fseek(f,0,SEEK_SET);
    u8 *buf = malloc(sz);
    if (fread(buf,1,sz,f) != sz) {
        fprintf(stderr, "Error: reading %s\n", privfile);
        fclose(f); free(buf); return -1;
    }
    fclose(f);

    ec_params params;
    import_params(&params, &secp256r1_str_params);
    ec_key_pair kp;
    if (ec_structured_priv_key_import_from_buf(
    &kp.priv_key, &params,
    buf, (u8)sz, ECDSA) != 0)
    {
        fprintf(stderr, "Error: invalid private key\n");
        free(buf); return -1;
    }
    free(buf);

    if (init_pubkey_from_privkey(&kp.pub_key, &kp.priv_key) != 0) {
        fprintf(stderr, "Error: deriving public key\n"); return -1;
    }

    /* Build entry content: timestamp|level|message */
    char ts[20];
    current_timestamp(ts, sizeof(ts));
```

```c
    size_t L = strlen(ts)+1+strlen(level)+1+strlen(message);
    char *entry = malloc(L+1);
    snprintf(entry, L+1, "%s|%s|%s", ts, level, message);

    /* Hash entry */
    const hash_mapping *hm;
    if (get_hash_by_type(SHA256, &hm) != 0) {
        fprintf(stderr, "Error: SHA256 unavailable\n");
        free(entry); return -1;
    }
    hash_context ctx;
    hm->hfunc_init(&ctx);
    hm->hfunc_update(&ctx, (u8*)entry, (u32)strlen(entry));
    u8 digest[64], dlen = hm->digest_size;
    hm->hfunc_finalize(&ctx, digest);

    /* Sign */
    u8 siglen;
    ec_get_sig_len(&params, ECDSA, SHA256, &siglen);
    u8 *sigbin = malloc(siglen);
    if (ec_sign(sigbin, siglen,
    &kp, digest, dlen,
    ECDSA, SHA256, NULL, 0) != 0)
    {
        fprintf(stderr, "Error: signing failed\n");
        free(sigbin); free(entry); return -1;
    }
    char *hex = to_hex(sigbin, siglen);
    free(sigbin);

    /* Append to log file */
    FILE *logf = fopen(logfile, "a");
    if (!logf) { perror(logfile); free(hex); free(entry); return -1; }
    fprintf(logf, "%s|%s\n", entry, hex);
    fclose(logf);

    printf("Appended log entry: [%s] %s\n", level, message);
    free(hex);
    free(entry);
    return 0;
}

/* Verify all entries in a log file */
static int do_verify(const char *pubfile, const char *logfile) {
    /* Load public key blob */
    FILE *f = fopen(pubfile, "rb");
    if (!f) { perror(pubfile); return -1; }
    fseek(f,0,SEEK_END);
    size_t sz = ftell(f);
    fseek(f,0,SEEK_SET);
    u8 *buf = malloc(sz);
    if (fread(buf,1,sz,f) != sz) {
        fprintf(stderr, "Error: reading %s\n", pubfile);
        fclose(f); free(buf); return -1;
    }
    fclose(f);

    ec_params params;
    import_params(&params, &secp256r1_str_params);
    ec_pub_key pub;
    if (ec_structured_pub_key_import_from_buf(&pub, &params, buf, (u8)sz, ECDSA) != 0)
    {
        fprintf(stderr, "Error: invalid public key\n");
        free(buf); return -1;
    }
    free(buf);

    /* Prepare for verification */
    const hash_mapping *hm;
    get_hash_by_type(SHA256, &hm);

    FILE *logf = fopen(logfile, "r");
    if (!logf) { perror(logfile); return -1; }

    char *line = NULL;
    size_t cap = 0;
    ssize_t len;
    int line_no = 0, bad = 0;
    while ((len = getline(&line, &cap, logf)) > 0) {
        line_no++;
        trim_nl(line);

        /* Split into timestamp|level|message|hexsig */
        char *ts = strtok(line, "|");
        char *lvl = strtok(NULL, "|");
        char *msg = strtok(NULL, "|");
        char *hex = strtok(NULL, "|");
        if (!ts||!lvl||!msg||!hex) {
            fprintf(stderr, "Format error on line %d\n", line_no);
            bad++;
            continue;
        }

        /* Rebuild content to verify */
        size_t L = strlen(ts)+1+strlen(lvl)+1+strlen(msg);
        char *entry = malloc(L+1);
        snprintf(entry, L+1, "%s|%s|%s", ts, lvl, msg);

        /* Hash */
        hash_context ctx;
        hm->hfunc_init(&ctx);
        hm->hfunc_update(&ctx, (u8*)entry, (u32)strlen(entry));
        u8 digest[64], dlen = hm->digest_size;
        hm->hfunc_finalize(&ctx, digest);

        /* Decode hexsig */
        size_t sl = strlen(hex)/2;
        u8 *sigbin = malloc(sl);
        for (size_t i = 0; i < sl; i++) {
            sscanf(hex + 2*i, "%2hhx", &sigbin[i]);
        }

        /* Verify */
        int rc = ec_verify(sigbin, (u8)sl,
        &pub, digest, dlen,
        ECDSA, SHA256, NULL, 0);
        if (rc != 0) {
            printf("Line %d: INVALID signature\n", line_no);
            bad++;
        }

        free(entry);
        free(sigbin);
    }
    free(line);
    fclose(logf);

    if (bad == 0) {
        printf("All entries verified successfully (%d lines).\n", line_no);
        return 0;
    } else {
        printf("%d of %d entries FAILED verification.\n", bad, line_no);
        return 1;
    }
}

int main(int argc, char **argv) {
    if (argc < 2) {
        usage(argv[0]);
        return 1;
    }
    if (!strcmp(argv[1], "generate") && argc == 3)
    return do_generate(argv[2]) ? 1 : 0;
    if (!strcmp(argv[1], "write") && argc >= 5) {
        /* Join message args */
        size_t mlen = 0;
        for (int i = 4; i < argc; i++) mlen += strlen(argv[i]) + 1;
        char *msg = malloc(mlen+1);
        msg[0] = '\0';
        for (int i = 4; i < argc; i++) {
            strcat(msg, argv[i]);
            if (i < argc-1) strcat(msg, " ");
        }
        int r = do_write(argv[2], argv[3], argv[4], msg);
        free(msg);
        return r ? 1 : 0;
    }
    if (!strcmp(argv[1], "verify") && argc == 4)
    return do_verify(argv[2], argv[3]) ? 1 : 0;

    usage(argv[0]);
    return 1;
}
```

```
gcc -std=c99 -Wall -Wextra -O2 -Iinclude -c src/logtool_ecc.c -o obj/logtool_ecc.o
gcc -std=c99 -Wall -Wextra -O2 -Iinclude -o bin/logtool_ecc \
      obj/logtool_ecc.o \
            lib/libarith.a \
            lib/libec.a \
            lib/libsign.a
```

logaggregator_ecc.c

```c
/*
* logaggregator_ecc.c
*
* Reads raw log lines from stdin, signs each line with real ECDSA (P-256),
* and writes them into rotating chunk files.
*
* Usage:
* logaggregator_ecc <privkey_file> <chunk_dir> [lines_per_chunk]
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdint.h>
#include <errno.h>

#include <crypto/cryptosig.h>

#define MAX_LINE 8192
#define DEFAULT_LINES_PER_CHUNK 1000000

/* Hex-encode */
static char *to_hex(const uint8_t *buf, size_t len) {
    static const char hex[] = "0123456789ABCDEF";
    char *out = malloc(len*2 + 1);
    for (size_t i = 0; i < len; i++) {
        out[2*i] = hex[buf[i] >> 4];
        out[2*i+1] = hex[buf[i] & 0xF];
    }
    out[len*2] = '\0';
    return out;
}

/* Trim newline */
static void trim_nl(char *s) {
    size_t n = strlen(s);
    while (n && (s[n-1]=='\r' || s[n-1]=='\n')) s[--n]='\0';
}

int main(int argc, char **argv) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s <privkey> <chunk_dir> [lines_per_chunk]\n",
            ↪ argv[0]);
        return 1;
    }
    const char *privfile = argv[1];
    const char *chunkdir = argv[2];
    long lines_per_chunk = DEFAULT_LINES_PER_CHUNK;
    if (argc == 4) lines_per_chunk = atol(argv[3]);

    /* Load P-256 params */
    // const ec_params *params = ec_maps[SECP256R1].params;
    ec_params params;
    import_params(&params, &secp256r1_str_params);

    /* Load private key blob */
    FILE *kf = fopen(privfile, "rb");
    if (!kf) { perror(privfile); return 1; }
    fseek(kf, 0, SEEK_END);
    size_t priv_sz = ftell(kf);
    fseek(kf, 0, SEEK_SET);
    uint8_t *priv_buf = malloc(priv_sz);
    // fread(priv_buf,1,priv_sz,kf);
    size_t got = fread(priv_buf, 1, priv_sz, kf);
    if (got != priv_sz) {
        if (feof(kf)) {
            fprintf(stderr, "Error: unexpected EOF reading private key\n");
            free(priv_buf);
            fclose(kf);
            return 1;
        } else {
            perror("fread");
            free(priv_buf);
        }
        fclose(kf);
        return -1;
    }
    fclose(kf);

    /* Import private key */
    ec_key_pair kp;
    if (ec_structured_priv_key_import_from_buf(
        &kp.priv_key, &params, priv_buf, (u8)priv_sz, ECDSA) != 0)
    {
        fprintf(stderr, "Error: invalid private key\n");return 1;
    }
    free(priv_buf);

    /* Derive public key */
    if (init_pubkey_from_privkey(&kp.pub_key, &kp.priv_key) != 0) {
        fprintf(stderr, "Error: deriving public key\n");
        return 1;
    }

    /* Prepare chunk files */
    if (mkdir(chunkdir, 0755) && errno!=EEXIST) {
        perror("mkdir chunk_dir"); return 1;
    }
    char fname[512];
    int chunk_seq = 1;
    snprintf(fname, sizeof(fname), "%s/chunk_%04d.log", chunkdir, chunk_seq);
    FILE *out = fopen(fname, "w");
    if (!out) { perror(fname); return 1; }
    fprintf(stderr, "Writing to %s\n", fname);

    /* Setup hash algorithm */
    const hash_mapping *hm;
    if (get_hash_by_type(SHA256, &hm) != 0) {
        fprintf(stderr, "Error: SHA256 not available\n"); return 1;
    }

    /* Process lines */
    char *line = NULL;
    size_t cap = 0;
    ssize_t len;
    long count = 0;
    while ((len = getline(&line, &cap, stdin)) > 0) {
        trim_nl(line);
        /* Hash the line */
        hash_context ctx;
        hm->hfunc_init(&ctx);
        hm->hfunc_update(&ctx, (uint8_t*)line, (u32)strlen(line));
        uint8_t digest[64];
        uint8_t dlen = hm->digest_size;
        hm->hfunc_finalize(&ctx, digest);

        /* Sign */
        u8 siglen;
        ec_get_sig_len(&params, ECDSA, SHA256, &siglen);
        uint8_t *sigbin = malloc(siglen);
        if (ec_sign(sigbin, siglen,
        &kp, digest, dlen,
        ECDSA, SHA256, NULL, 0) != 0)
        {
            fprintf(stderr, "Error: signing failed\n");
            free(sigbin);
            break;
        }
        char *hex = to_hex(sigbin, siglen);
        free(sigbin);

        /* Write signed line */
        fprintf(out, "%s|%s\n", line, hex);
        fflush(out);
        free(hex);

        if (++count >= lines_per_chunk) {
            fclose(out);
            count = 0;
            chunk_seq++;
            snprintf(fname, sizeof(fname), "%s/chunk_%04d.log", chunkdir, chunk_seq);
            out = fopen(fname, "w");
            if (!out) { perror(fname); break; }
            fprintf(stderr, "Writing to %s\n", fname);
        }
    }
    free(line);
    fclose(out);
    fprintf(stderr, "Done: %d chunk(s)\n", chunk_seq);
    return 0;
}
```

```
gcc -std=c99 -Wall -Wextra -O2 -Iinclude -c src/logaggregator_ecc.c -o obj/logaggregator_ecc.o
gcc -std=c99 -Wall -Wextra -O2 -Iinclude -o bin/logaggregator_ecc \
      obj/logaggregator_ecc.o \
            lib/libarith.a \
            lib/libec.a \
            lib/libsign.a
```

generate_logs.py

```python
"""
generate_logs.py

Generate a large synthetic log file with timestamp|level|message entries.
Usage:
python3 generate_logs.py --lines 10000 --outfile app.log
"""

import argparse
import random
from datetime import datetime

LEVELS = ['INFO', 'WARN', 'ERROR', 'DEBUG']
MESSAGES = [
    'User login successful',
    'File processed',
    'Connection timeout',
    'Data saved to DB',
    'Error reading configuration',
    'Heartbeat',
    'Cache miss',
    'Session expired'
]

def main():
    p = argparse.ArgumentParser(description="Generate synthetic log lines")
    p.add_argument('--lines', type=int, default=1_000_000, help='Number of lines to
        ↪ generate')
    p.add_argument('--outfile', type=str, default='app.log', help='Output log
        ↪ filename')
    args = p.parse_args()

    with open(args.outfile, 'w') as f:
        for i in range(args.lines):
            ts = datetime.utcnow().strftime('%Y-%m-%d %H:%M:%S')
            level = random.choice(LEVELS)
            msg = random.choice(MESSAGES)
            f.write(f'{ts}|{level}|{msg}\n')

    print(f"Generated {args.lines} lines in '{args.outfile}'")

if __name__ == '__main__':
    main()
```

```
python3 generate_logs.py --lines 10000 --outfile app.log
```

6.2 Example Scenario with `logtool_ecc`

```
content...
```

6.3 Example Scenario with `logaggregator_ecc`

```
content...
```

Bibliography

[BEF17] Ryad Benadjila, Arnaud Ebalard, and Jean-Pierre Flori. *libecc: A lightweight and built-in ECC library*. GitHub repository, 2017. `https://github.com/libecc/libecc`, (Accessed: June 10, 2025).

[CMR+23] Lily Chen, Dustin Moody, Andrew Regenscheid, et al. *NIST Special Publication 800-186, Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters*. U.S. Department of Commerce, Gaithersburg, MD, February 2023.

[NIST13] National Institute of Standards and Technology. *FIPS PUB 186-4, Digital Signature Standard (DSS).* U.S. Department of Commerce, Gaithersburg, MD, July 2013.

[NIST23] National Institute of Standards and Technology. *FIPS PUB 186-5, Digital Signature Standard (DSS).* U.S. Department of Commerce, Gaithersburg, MD, February 2023.

[Por13] Thomas Pornin. *RFC 6979: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA).* Internet Engineering Task Force, August 2013.

Appendix A

Integrated Makefile for Applications

```
├─── include
│    └─── crypto
│         ├─── ...
├─── lib
│    ├─── libarith.a
│    ├─── libec.a
│    └─── libsign.a
├─── src
│    ├─── filecheck_ecc.c
│    ├─── generate_file.py
│    ├─── generate_logs.py
│    ├─── logaggregator_ecc.c
│    └─── logtool_ecc.c
│
```

Build without Makefile

```
mkdir -p obj bin
gcc -std=c99 -Wall -Wextra -O2 -Iinclude -c src/filecheck_ecc.c -o obj/filecheck_ecc.o
gcc -std=c99 -Wall -Wextra -O2 -Iinclude -o bin/filecheck_ecc obj/filecheck_ecc.o \
       lib/libarith.a \
       lib/libec.a \
       lib/libsign.a

gcc -std=c99 -Wall -Wextra -O2 -Iinclude -c src/logtool_ecc.c -o obj/logtool_ecc.o
gcc -std=c99 -Wall -Wextra -O2 -Iinclude -o bin/logtool_ecc \
       obj/logtool_ecc.o \
       lib/libarith.a \
       lib/libec.a \
       lib/libsign.a

gcc -std=c99 -Wall -Wextra -O2 -Iinclude -c src/logaggregator_ecc.c -o obj/logaggregator_ecc.o
gcc -std=c99 -Wall -Wextra -O2 -Iinclude -o bin/logaggregator_ecc \
       obj/logaggregator_ecc.o \
       lib/libarith.a \
       lib/libec.a \
       lib/libsign.a
```

Build with Makefile

```
CC = gcc
CFLAGS = -std=c99 -Wall -Wextra -O2 -Iinclude

SRCDIR = src
OBJDIR = obj
BINDIR = bin
LIBDIR = lib

# Your tools
TOOLS = filecheck_ecc logtool_ecc logaggregator_ecc

# Object files
OBJS = \
	$(OBJDIR)/common_utils.o \
	$(OBJDIR)/filecheck_ecc.o \
	$(OBJDIR)/logtool_ecc.o \
	$(OBJDIR)/logaggregator.o \

all: prep $(addprefix $(BINDIR)/, $(TOOLS))

prep:
	mkdir -p $(OBJDIR) $(BINDIR)

# Compile sources
$(OBJDIR)/%.o: $(SRCDIR)/%.c
	$(CC) $(CFLAGS) -c $< -o $@

# Link filecheck_ecc
$(BINDIR)/filecheck_ecc: $(OBJDIR)/filecheck_ecc.o
	$(CC) $(CFLAGS) -o $@ $^ \
		$(LIBDIR)/libarith.a $(LIBDIR)/libec.a $(LIBDIR)/libsign.a

# Link logtool_ecc
$(BINDIR)/logtool_ecc: $(OBJDIR)/logtool_ecc.o
	$(CC) $(CFLAGS) -o $@ \
		$^ \
		$(LIBDIR)/libarith.a \
		$(LIBDIR)/libec.a \
		$(LIBDIR)/libsign.a

# Link logaggregator (if it uses ECDSA too)
$(BINDIR)/logaggregator_ecc: $(OBJDIR)/logaggregator_ecc.o
	$(CC) $(CFLAGS) -o $@ \
		$^ \
		$(LIBDIR)/libarith.a \
		$(LIBDIR)/libec.a \
		$(LIBDIR)/libsign.a

clean:
rm -rf $(OBJDIR) $(BINDIR)

.PHONY: all prep clean
```