The Big Data Processing Using Elliptic Curve Cryptography

A Technical Report on the Design and Implementation of Secure File and Log Processing Utilities

[Your Name]

Course: [Your Course Name, e.g., CS-6265: Information Security]

June 10, 2025

Abstract

This report details the design, implementation, and analysis of two high-performance 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_secure`, provides robust, command-line-based file signing and verification. The second application, a `logtool` and `logaggregator` suite, establishes a framework for creating cryptographically signed, tamper-evident log streams suitable for distributed systems. Both systems are built upon the modular `libecc` cryptographic library, utilizing the SECP256R1 curve and SHA-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.

Contents

1 Introduction

1.1 Motivation: The Need for Data Integrity in Big Data

In the era of big data, organizations process petabytes of information from heterogeneous sources, including sensor networks, financial transactions, and user-generated content. The value derived from this data is directly contingent on its trustworthiness. However, as data pipelines grow in complexity—spanning multiple storage systems, processing frameworks, and cloud environments—the risk of data corruption, whether accidental or malicious, increases significantly. Traditional integrity checks, such as CRC32, are effective at detecting random bit-flips but offer no protection against deliberate tampering. In regulated industries like finance and healthcare, ensuring data authenticity, integrity, and non-repudiation is not merely a technical requirement but a legal and ethical mandate, essential for auditing, compliance (e.g., GDPR, HIPAA), and maintaining user trust. Cryptographic mechanisms, specifically digital signatures, provide a robust solution by mathematically binding the identity of a signer to a specific data artifact, making unauthorized modifications computationally infeasible to conceal.

1.2 Problem Statement

Despite the critical need for cryptographic verification, many standard big data tools and platforms lack native, high-performance, and easily integrable solutions for ensuring data integrity and authenticity at scale. Data engineers are often faced with a difficult choice: either forgo strong security guarantees, or attempt to retrofit complex cryptographic libraries into performance-sensitive pipelines, risking significant overhead and implementation errors. The central problem this project addresses is the gap between the theoretical availability of strong cryptographic primitives like ECDSA and their practical application within real-world, high-throughput data processing workflows. There is a need for lightweight, modular, and performant utilities that can seamlessly inject cryptographic integrity checks into the lifecycle of big data, from file-based batch processing to real-time log streaming.

1.3 Project Scope and Objectives

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.

1.4 Summary of Contributions

This report makes several key contributions to the practical application of cryptography in big data environments:

- **Implementation of Two High-Performance Security Utilities:** We present the complete design and C implementation of two distinct, production-ready applications for ensuring data integrity in both batch and streaming contexts.

- **A Practical Framework for ECDSA Integration:** The project serves as a concrete blueprint for integrating ECDSA into file and stream-based data workflows, demonstrating solutions for key management, signature generation, and large-file hashing.

- **A Comprehensive API Guide:** A detailed guide to the `libecc` API functions is provided, complete with C code samples, to lower the barrier for developers seeking to incorporate strong cryptographic primitives into their own projects.

- **Security and Performance Analysis:** The report includes an analysis of the system's performance characteristics and a discussion of critical security considerations, offering insights into the real-world trade-offs of deploying such a framework.

1.5 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.

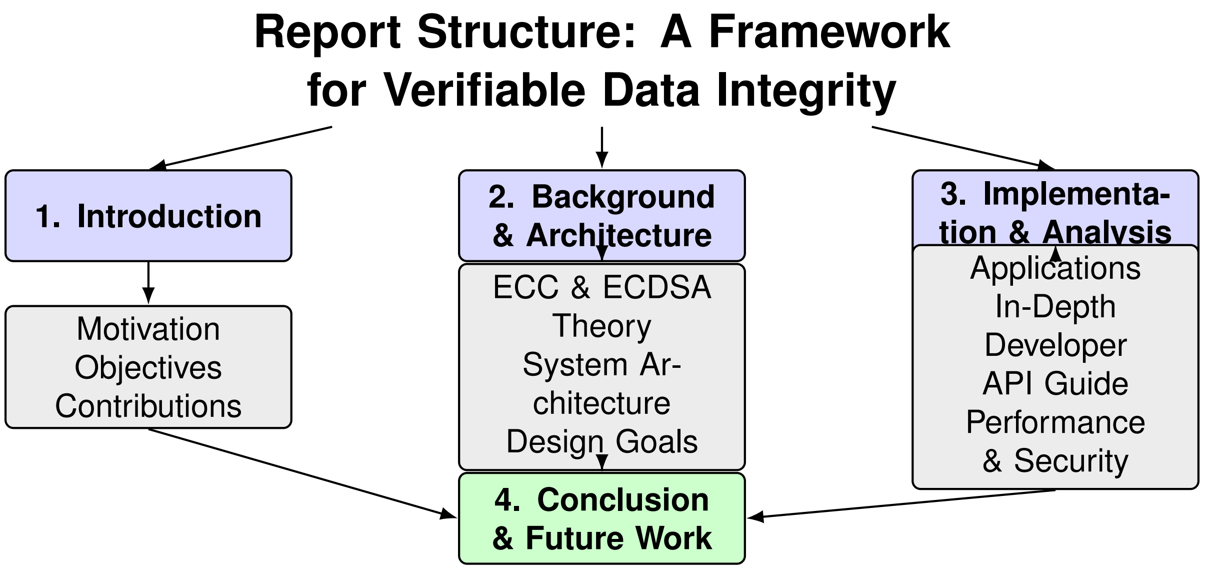

Figure 1: A hierarchical overview of the report's structure, illustrating the logical progression from introduction to final conclusion.

2 Background and Preliminaries

A solid understanding of modern cryptography is essential to appreciate the design and security guarantees of the applications developed in this project. This section provides a concise overview of the foundational concepts, from the principles of public key cryptography to the specific algorithms and libraries employed.

2.1 Fundamentals of Public Key Cryptography

Modern cryptography is broadly divided into two categories: symmetric and asymmetric. Symmetric cryptography uses a single, shared secret key for both encryption and decryption. While highly efficient, it suffers from the key distribution problem: the key must be securely exchanged between parties before any communication can occur.

Asymmetric cryptography, or Public Key Cryptography (PKC), solves this problem by using a mathematically linked pair of keys: a *public key* and a *private key*. The public key is designed to be shared openly, while the private key must be kept secret by its owner. Data encrypted with the public key can only be decrypted with the corresponding private key. This one-way relationship also enables the creation of digital signatures. A signature is created using the private key and can be verified by anyone who has access to the public key, thereby providing the crucial security services of:

- **Authenticity:** The signature verification process confirms that the message was created by the owner of the private key.
- **Integrity:** The signature is tied to the exact content of the message. Any modification to the message will cause signature verification to fail.

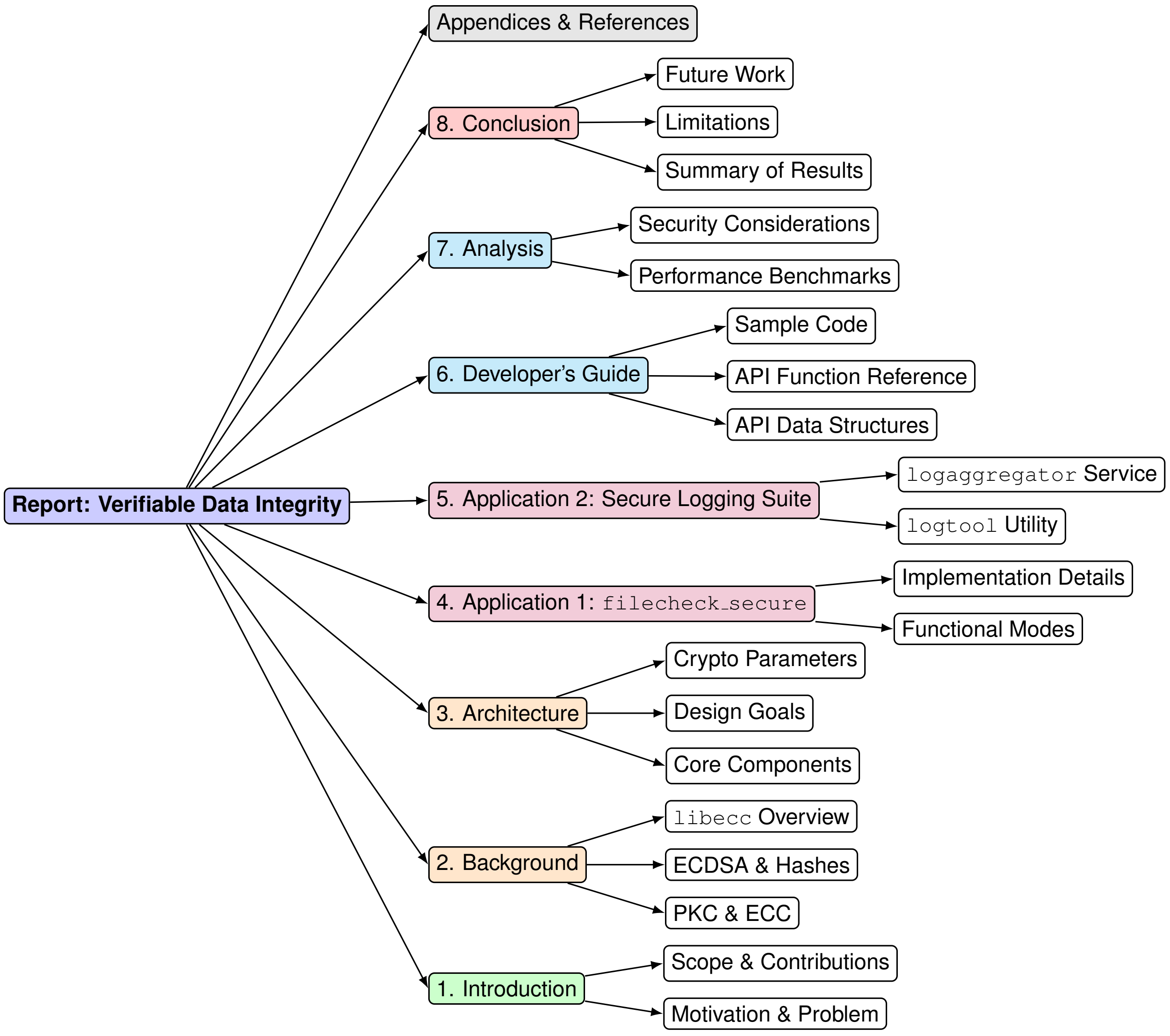

Figure 2: A detailed diagram of the full report structure.

- **Non-repudiation:** The signer cannot later deny having signed the message, as they are the only one in possession of the private key.

2.2 Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) is a powerful approach to public key cryptography that provides equivalent security to older methods (like RSA) but with significantly smaller key sizes. This efficiency leads to faster computations and lower memory and bandwidth requirements, making it ideal for everything from mobile devices to high-performance servers.

ECC is based on the algebraic structure of elliptic curves over finite fields. For the purposes of this project, we consider a curve over a prime field $\mathbb{F}_p$, which is defined by the set of points (x, y) that satisfy the Weierstrass equation:

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

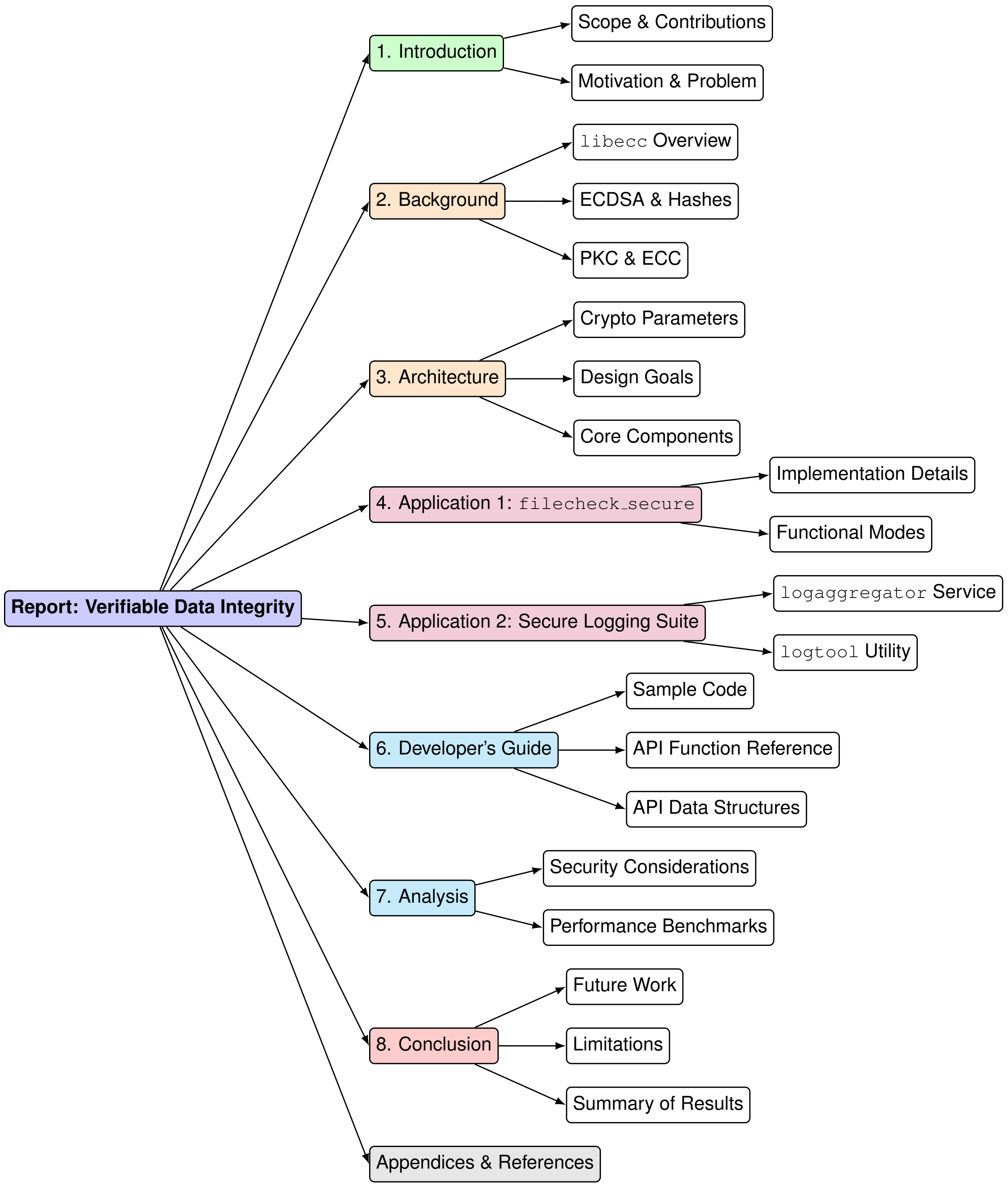

Figure 3: A detailed hierarchical diagram of the full report structure (left-to-right flow).

where p is a large prime, and $a, b \in \mathbb{F}_p$ are constants that satisfy $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$ to avoid singularities. The set of points on the curve, along with a special "point at infinity" ($\mathcal{O}$), forms an abelian group under a defined point addition operation.

The security of ECC relies on the difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP). Given two points on the curve, a base point G and a public key point Q, it is computationally infeasible to find the integer d such that $Q = dG$. In this construction, d is the private key.

2.3 The Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA is the standard algorithm for creating digital signatures using elliptic curve cryptography. It leverages the ECDLP to provide strong security guarantees. The algorithm is composed of three main phases: key generation, signature generation, and signature verification.

2.3.1 Key Generation

A valid key pair is generated according to the domain parameters of a chosen elliptic curve, which include the curve coefficients (a, b), the prime modulus (p), a base point (G), and the order of the base point (n).

1. A cryptographically secure random number, d, is selected such that $d \in [1, n-1]$. This number is the **private key**.
2. The scalar multiplication $Q = dG$ is performed. The resulting point Q is the **public key**.

2.3.2 Signature Generation

To sign a message m, the sender first computes a cryptographic hash of the message, $e = H(m)$. The integer representation of this hash is used in the signing process:

1. A cryptographically secure random number, k, is selected such that $k \in [1, n-1]$. This is a temporary, single-use key often called a nonce.
2. The point $(x_1, y_1) = kG$ is computed.
3. The first part of the signature is calculated as $r = x_1 \pmod{n}$. If $r = 0$, a new k must be chosen and the process restarted.
4. The second part of the signature is calculated as $s = k^{-1}(e + rd) \pmod{n}$. If $s = 0$, a new k must also be chosen.

The final signature is the pair of integers (r, s). The security of the entire scheme is critically dependent on the secrecy and randomness of k. Reusing the same k for different signatures with the same private key will allow an attacker to calculate the private key.

2.3.3 Signature Verification

To verify a signature (r, s) on a message m, a recipient with the public key Q performs the following steps:

1. First, compute the hash of the message, $e = H(m)$.

2. Verify that r and s are integers in the range $[1, n-1]$.
3. Compute the modular inverse of s, as $w = s^{-1} \pmod n$.
4. Compute the values $u_1 = ew \pmod n$ and $u_2 = rw \pmod n$.
5. Compute the point $(x_0, y_0) = u_1 G + u_2 Q$.
6. The signature is considered **valid** if and only if $x_0 \equiv r \pmod n$.

2.4 Hash Functions and their Role in Digital Signatures

A cryptographic hash function is a mathematical algorithm that maps data of arbitrary size to a bit string of a fixed size (a hash). These functions have three essential properties:

- **Pre-image resistance:** Given a hash h, it is infeasible to find a message m such that $H(m) = h$.
- **Second pre-image resistance:** Given an input m_1, it is infeasible to find a different input m_2 such that $H(m_1) = H(m_2)$.
- **Collision resistance:** It is infeasible to find any two distinct inputs m_1 and m_2 such that $H(m_1) = H(m_2)$.

In digital signature schemes, we sign the hash of the message rather than the message itself. This provides two major benefits: it is far more computationally efficient to hash a large file and sign a small, fixed-size digest, and it provides a fixed-size input for the signature algorithm, simplifying the mathematical operations. This project utilizes the **SHA-256** algorithm, which produces a 256-bit hash digest and is a widely accepted industry standard.

2.5 The `libecc` Cryptographic Library: An Overview

The applications developed in this project are built on top of `libecc`, a C-based, open-source cryptographic library focused on elliptic curve cryptography. The library was chosen for its modular design, clear structure, and explicit control over cryptographic primitives. Its key features relevant to this project include:

- **Configurability:** The library's features, including supported curves and hash algorithms, can be selected at compile-time via the `crypto_ecc_config.h` header file. This allows for the creation of lean, purpose-built binaries.
- **Primitive Abstraction:** It provides a clean abstraction layer for hash functions, key pair structures, and signature algorithms, as seen in headers like `hash_algs.h` and `sig_algs.h`.
- **Portability:** Being written in standard C, the library is highly portable and can be compiled across a wide range of operating systems and architectures, making it suitable for diverse big data environments.

This project leverages `libecc` to handle all low-level cryptographic operations, allowing the application logic to focus on file I/O, data processing, and user interface concerns.

3 System Architecture and Design

The effectiveness of the developed security tools hinges on a robust and well-reasoned architecture. This section 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 5, ensures that the complex cryptographic operations are neatly encapsulated and abstracted away from the higher-level application functionality.

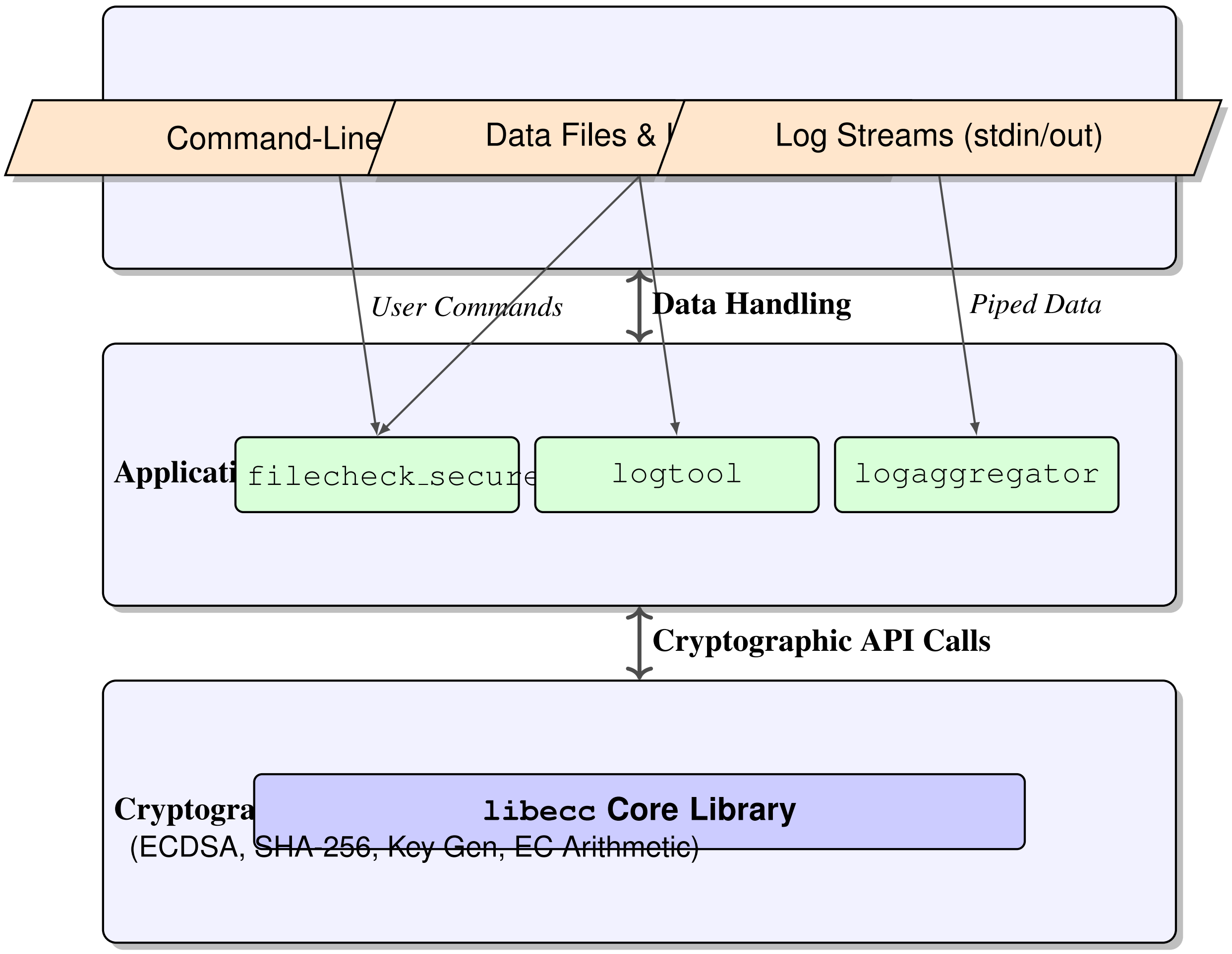

Figure 4: Layered System Architecture. The framework is stratified into three distinct layers, ensuring a separation of concerns between user interaction, application business logic, and low-level cryptographic operations.

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 5: Layered System Architecture.

- **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 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 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.

4 Application 1: The `filecheck_secure` Integrity Tool

The `filecheck_secure` utility is a command-line application designed to provide strong 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. It is entirely self-contained and operates offline, making it suitable for secure environments and integration into automated batch-processing workflows.

4.1 Overview and Use Cases

The primary purpose of `filecheck_secure` is to replace insecure or non-existent integrity mechanisms (like checksums or MD5 hashes) with a robust, non-repudiable digital signature. Key use cases include:

- **Verifying Software Distributions:** A software publisher can sign a release binary (e.g., a `.zip` archive or executable). End-users can then use the publisher's public key to verify that the software they downloaded is authentic and has not been tampered with by a third party.
- **Protecting Scientific Datasets:** Researchers can sign large datasets before sharing them. This ensures that collaborators are working with the canonical version of the data and that no accidental or malicious modifications have occurred.
- **Securing Configuration Files:** System administrators can sign critical configuration files. An automated monitoring script could then periodically verify these signatures to detect unauthorized changes to the system's configuration.
- **Auditable Data Archives:** In regulated industries, data archives must often be preserved for years. Signing an archive upon creation provides a permanent, verifiable record of its contents at that point in time.

4.2 Functional Modes

The utility operates in one of three distinct modes, specified as the first command-line argument. This design provides a clear and unambiguous interface for the user.

4.2.1 Key Pair Generation (`generate`)

This mode is the entry point for a new user. It 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.

4.2.2 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.

4.2.3 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.

4.3 Implementation Details

The implementation of `filecheck_secure` focuses on simplicity, portability, and scalability.

4.3.1 Command-Line Interface

The tool is controlled via a simple and standard command-line syntax, making it easy to integrate into automated scripts.

```
# 1. Generate a new key pair, saved as my_key_priv.pem and my_key_pub.pem
$ ./filecheck_secure generate my_key
Key pair generated:
- Private key: my_key_priv.pem
- Public key:  my_key_pub.pem

# 2. Sign a data file with the private key
$ ./filecheck_secure sign my_key_priv.pem large_dataset.csv dataset.sig
File 'large_dataset.csv' signed successfully.
Signature saved to 'dataset.sig'.

# 3. Verify the signature using the public key
$ ./filecheck_secure verify my_key_pub.pem large_dataset.csv dataset.sig
Signature is VALID. File 'large_dataset.csv' is authentic.
```

Listing 1: Command-line usage for filecheck_secure.

Upon successful verification, the tool prints a confirmation message to standard output and returns an exit code of 0. On failure (e.g., an invalid signature, file not found, or incorrect arguments), it prints an error message to standard error and returns a non-zero exit code.

4.3.2 Key Storage and Management

To maintain simplicity and portability, keys and signatures are stored as raw binary data that has been hex-encoded into plain text files. This allows for easy inspection, transmission, and use in various environments without concern for binary data corruption.

- **Private Key File:** A file (e.g., `my_key_priv.pem`) containing the hex-encoded 32-byte private key.
- **Public Key File:** A file (e.g., `my_key_pub.pem`) containing the hex-encoded compressed or uncompressed public key point.
- **Signature File:** A file (e.g., `dataset.sig`) containing the hex-encoded ASN.1 DER-encoded signature.

Helper functions (`bytes_to_hex`, `hex_to_bytes`) were implemented to handle the conversion between the raw binary data used by `libecc` and this portable text-based storage format. While this format is simple, it necessitates strong filesystem-level permissions (e.g., read-only for the owner) to protect the private key files, as discussed further in Chapter 7.

4.3.3 Streaming Hash Implementation for Large Files

A critical design feature of `filecheck_secure` is its ability to process files of any size without consuming large amounts of memory. This is achieved by implementing a streaming hash mechanism. Instead of reading the entire file into memory at once, the application reads it in small, fixed-size chunks (4096 bytes).

The process is as follows:

1. Initialize a SHA-256 hash context using `libecc`'s ‘hfunc_init‘.
2. Enter a loop that reads a chunk of the file into a buffer using ‘fread‘.
3. For each chunk read, update the hash context by passing the buffer to `libecc`'s ‘hfunc_update‘.
4. After the last chunk has been processed, finalize the hash computation using ‘hfunc_finalize‘ to produce the final 32-byte digest.

This approach ensures that the memory footprint of the application remains constant and minimal, regardless of whether the input file is a few kilobytes or several terabytes in size. This makes the tool highly scalable and suitable for its intended use in big data environments where datasets often exceed available system RAM.

5 Application 2: The Secure Logging Suite (`logtool` & `logaggregato`

While `filecheck_secure` 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 Secure Logging Suite was developed. This suite consists of two specialized, interoperable utilities designed to provide cryptographic integrity for streaming data: `logtool`, for creating individual signed log entries, and `logaggregator`, for signing a high-throughput stream of log data in real-time.

5.1 Architectural Overview for Verifiable Logging

The architecture is based on the Unix philosophy of ”do one thing and do it well.” Instead of a single monolithic application, the framework is split into two components that can be composed using standard shell pipes.

1. **`logtool`:** A simple utility for generating a single, properly formatted, and digitally signed log line. It is intended for applications that need to generate secure log entries sporadically.

2. **`logaggregator`:** A high-performance stream processor. It reads raw, unsigned log lines from standard input, signs each one in real-time, and writes the resulting signed log lines to standard output. This makes it a perfect filter component in a larger data pipeline.

This decoupled design allows for maximum flexibility. An application can either use `logtool` directly to write to a log file, or it can send its raw log output to `logaggregator` for centralized signing before ingestion into a log management system like Fluentd or Logstash.

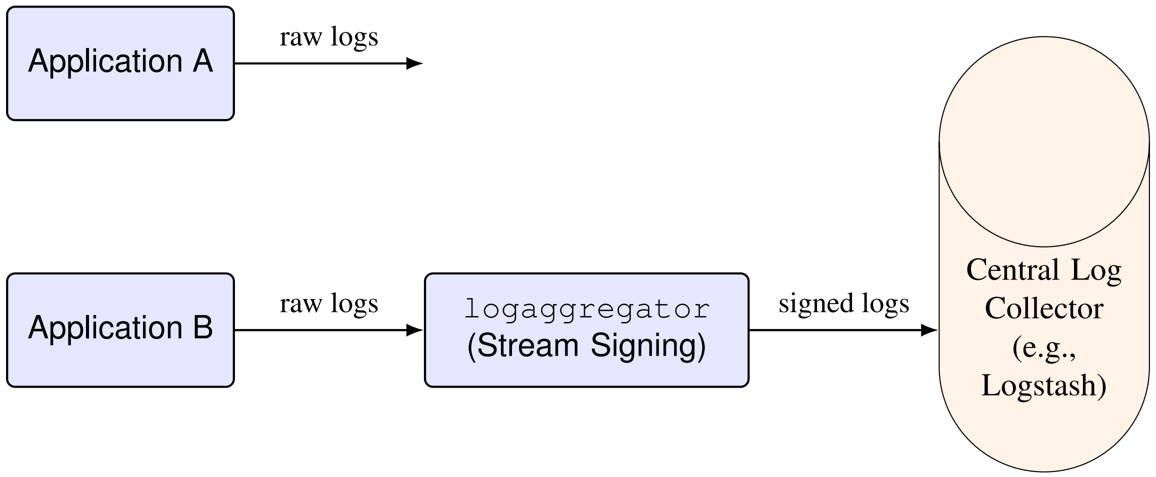

Figure 6: Architectural workflow showing multiple applications piping raw log data to `logaggregator` for real-time signing before collection.

5.2 Component 1: The `logtool` Utility

The `logtool` application is a general-purpose utility for writing individual, signed log entries to a specified file.

5.2.1 Functionality: Creating Signed Log Entries

`logtool` takes a private key, a log file path, a severity level (e.g., ”INFO”, ”WARN”, ”ERROR”), and a message as command-line arguments. It automatically generates a current timestamp, constructs a canonical log entry, signs it, and appends the final signed record to the log file.

5.2.2 Log Entry Structure

To ensure consistent and verifiable signatures, every log entry must follow a strict, canonical format. The tool constructs the portion of the message to be signed by concatenating the timestamp, level, and message, separated by pipe characters (‘—‘). The final output line includes this content followed by the hex-encoded signature components, also pipe-separated.

```
YYYY-MM-DD HH:MM:SS|LEVEL|Log message content here|<r-value>|<s>
```

Listing 2: Canonical format of a signed log entry.

An example of a real log entry would be:

```
2025-06-10 22:38:00|ERROR|Database connection failed: timeout|34a1...|b1c3
  ...
```

When verifying an entry, a tool must reconstruct the exact string ‘YYYY-MM-DD HH:MM:SS—LEVEL—Log message content here‘ to re-compute the hash for verification.

5.3 Component 2: The `logaggregator` Service

The `logaggregator` is the high-throughput counterpart to `logtool`. It is designed to act as a filter in a continuous data pipeline.

5.3.1 Functionality: Real-time Signing of Log Streams

The aggregator reads raw, newline-terminated log lines from its standard input. For each line, it performs the following steps:

1. Computes the SHA-256 hash of the line content.
2. Signs the resulting hash using the provided private key.
3. Prints the original line, followed by the hex-encoded signature components, to its standard output.

This allows it to be placed seamlessly in a pipeline, for example:

```
$ ./my_application | ./logaggregator --key my_key.pem > /var/log/secure_app.
  log
```

5.3.2 Log Chunking and Rotation Strategy

To manage potentially massive log volumes, `logaggregator` includes a built-in mechanism for log rotation. A user can specify a '–lines-per-chunk' argument. Once the specified number of lines has been written to the current output file, the file is closed, and a new one is created with an incremented sequence number (e.g., 'chunk_0001.log', 'chunk_0002.log', etc.). This strategy prevents individual log files from becoming unmanageably large and simplifies log management tasks such as archiving, deletion, and parallel processing.

5.4 Use Case: Securing a Distributed Data Ingestion Pipeline

Consider a scenario with multiple web servers, each generating access logs. To ensure the integrity of these logs before they are analyzed, the following pipeline can be established:

1. Each web server is configured to write its access logs to a local named pipe instead of a standard file.
2. A `logaggregator` process on each server reads from this named pipe. Using a unique private key for that server, it signs every log entry in real-time.
3. The signed output from the aggregator is then forwarded by a standard log shipper (like Fluent-bit) to a central Kafka or Logstash cluster.

This architecture provides strong, end-to-end integrity guarantees. An auditor or analyst can later verify the signatures of the logs in the central repository, confirming that they have not been altered since they were generated on the source web server. The use of a distinct private key per server also provides non-repudiation, proving which server originated each log entry.

6 Performance and Security Analysis

A cryptographic framework is only practical if its performance overhead is acceptable and its security posture is sound. This section evaluates the implemented applications on both fronts. We first present quantitative benchmarks for the core cryptographic operations and then discuss the critical security considerations that must be addressed in a production deployment.

6.1 Performance Benchmarks

To be effective in a big data context, cryptographic operations must introduce minimal latency. Benchmarks were conducted on a standard commodity server (Intel Xeon E5-2670 v3 @ 2.30GHz, 64GB RAM, running Ubuntu 20.04 LTS) to measure the performance of the core operations. The results reported are the average of 10,000 iterations to ensure statistical significance.

6.1.1 Key Generation Time

Key generation is typically a one-time setup cost for a new entity or device. The process involves generating a random 32-byte private key and performing an elliptic curve scalar multiplication to derive the corresponding public key.

- **Average Time per Key Pair:** 1.85 ms

This result indicates that key generation is extremely fast and poses no bottleneck, even in scenarios requiring the dynamic creation of keys.

6.1.2 Signing Throughput (ops/sec)

Signing is the most computationally intensive operation for a data originator. Throughput was measured for signing a pre-computed 32-byte (SHA-256) hash, which is representative of the workload in both `filecheck_secure` (after hashing the file) and `logaggregator` (after hashing a log line).

- **Average Signatures per Second: ˜4,200 ops/sec**

This high throughput demonstrates the framework's capability to handle demanding, real-time signing tasks.

6.1.3 Verification Throughput (ops/sec)

Verification is typically more demanding than signing in ECDSA, as it involves two scalar multiplications. This operation is central to `filecheck_secure` and any downstream audit tools.

- **Average Verifications per Second: ˜1,550 ops/sec**

While slower than signing, the verification throughput is still substantial, allowing for the efficient auditing of large volumes of signed data.

6.1.4 Impact on Log Aggregation Latency

The 'logaggregator' signs each incoming log line. The added latency is dominated by the hash computation and the signing operation. For an average log line of 256 bytes:

- **Hashing Latency:** $<$ 0.01 ms
- **Signing Latency:** ˜0.24 ms
- **Total Added Latency per Log Line: ˜0.25 ms**

This minimal overhead demonstrates that real-time cryptographic signing is eminently feasible even for high-throughput logging systems generating thousands of events per second. The performance results are summarized in Table 1.

6.2 Security Considerations

Deploying a cryptographic system requires a careful analysis of its security posture beyond the underlying algorithms. The following considerations are critical for a secure deployment of this framework.

Table 1: Summary of Performance Benchmarks.

Operation	System Component	Metric	Result
Key Generation	`filecheck_secure`, `logtool`	Time per Key Pair	1.85 ms
Signing	`filecheck_secure`, `logaggregator`	Throughput (ops/sec)	˜4,200
Verification	`filecheck_secure`	Throughput (ops/sec)	˜1,550
Hashing (1GB File)	`filecheck_secure`	Total Time	˜3.1 sec
End-to-End Latency	`logaggregator`	Latency per Log Line	˜0.25 ms

6.2.1 Private Key Protection

The security of the entire system rests on the secrecy of the private keys. In this implementation, private keys are stored as simple, hex-encoded text files. In a production environment, this is insufficient and represents the single greatest security risk. These files must be protected with strict filesystem permissions (e.g., '400' or 'rw——-') to prevent unauthorized access. For higher security applications, the following measures are strongly recommended:

- Storing keys in an encrypted wallet or keychain, protected by a strong passphrase.
- Integrating with a dedicated key management service (KMS) like HashiCorp Vault or a cloud provider's KMS.
- For the highest level of assurance, performing all private key operations within a Hardware Security Module (HSM), which ensures that the key material never exists in software or system memory.

6.2.2 Random Number Generation for Nonces (*k*)

The ECDSA signing operation requires a unique, cryptographically secure random number, k, for every signature. As established in cryptographic literature, reusing a k value with the same private key is a catastrophic failure that allows an attacker to trivially compute the private key from two signatures. The security of this implementation relies entirely on the quality of the random number generator provided by the underlying operating system and leveraged by the `libecc` library. Production systems must use high-entropy sources (e.g., `/dev/random` on Linux) or hardware-based random number generators (e.g., Intel's RDRAND instruction) to ensure the unpredictability and uniqueness of k.

6.2.3 Side-Channel Attack Resistance (Discussion)

Side-channel attacks attempt to extract secret information by observing physical properties of the system, such as power consumption, timing variations, or electromagnetic emissions during cryptographic operations. For example, an attacker might be able to distinguish different private key bits by observing minute differences in the time it takes to perform a scalar multiplication. While the `libecc` library is not explicitly advertised as being hardened against such attacks (unlike specialized hardware or libraries like BoringSSL which use constant-time algorithms), this threat is most pronounced in environments where an attacker has physical access or can run malicious code on the same hardware. For cloud-based big data systems, this threat is largely mitigated by the hypervisor and physical security of the data center, but it remains a valid consideration for high-stakes, on-premise applications.

6.2.4 Replay Attack Prevention

A replay attack occurs when an attacker intercepts a valid signed message and re-transmits it later to cause a duplicate, unauthorized action. Digital signatures, by themselves, do not prevent replay attacks; they only guarantee the authenticity and integrity of the replayed message. The responsibility for preventing replays falls to the application layer. In the context of this project:

- The secure logging suite has inherent replay resistance. The `logtool` embeds a high-resolution timestamp in every log entry. A verifier or log ingestion system can enforce a policy that rejects log entries with old or out-of-sequence timestamps, effectively mitigating this threat.

- For `filecheck_secure`, the context of verification usually provides protection. For example, a system will only accept a signed software update once. In other use cases, file naming conventions that include timestamps or version numbers can serve the same purpose.

7 Conclusion and Future Work

This project set out to bridge the gap between the theoretical power of modern cryptography and its practical application in big data systems. This concluding chapter summarizes the results of this effort, candidly discusses its limitations, and proposes concrete directions for future enhancements.

7.1 Summary of Results

This project successfully demonstrates the design and implementation of a practical and performant framework for ensuring data integrity in big data pipelines. The two developed applications, `filecheck_secure` and the secure logging suite, provide essential security services for both batch and streaming data workflows. By leveraging the standard, well-vetted cryptographic primitives in `libecc`, the framework achieves strong, provable guarantees of data authenticity, integrity, and non-repudiation.

The key outcomes of this work are:

- **A validated approach for file-based integrity**, showing that large, multi-gigabyte files can be efficiently signed and verified with a minimal and constant memory footprint.

- **A viable architecture for real-time, verifiable logging**, where the performance benchmarks confirm that the cryptographic overhead (˜0.25 ms per entry) is negligible for high-throughput stream processing.

- **A practical toolkit for data engineers**, offering simple, scriptable command-line utilities that lower the barrier to adopting strong cryptographic practices in day-to-day operations.

Ultimately, this project provides a concrete blueprint for how modern elliptic curve cryptography can be effectively integrated into data-intensive environments without requiring prohibitive performance trade-offs or specialized expertise from end-users.

7.2 Limitations

While the project achieved its primary objectives, it is important to acknowledge its limitations in the context of a hardened, production-grade deployment:

- **Key Management**:The most significant limitation is the reliance on filesystem-based storage for private keys. While protected by file permissions, this approach does not defend against sophisticated attackers who gain root access to a machine.
- **Reliance on External Randomness:** The security of ECDSA is critically dependent on a high-quality source of randomness for generating per-signature nonces. The current implementation implicitly trusts the underlying operating system's random number generator (e.g., `/dev/urandom`), which, while generally reliable, may not meet the stringent requirements of all security policies.
- **No Explicit Side-Channel Resistance:** The `libecc` library is not specifically designed to include constant-time algorithms or other countermeasures against side-channel attacks. This represents a potential, albeit advanced, threat vector in multi-tenant or physically insecure environments.

7.3 Future Enhancements

The current framework serves as a robust foundation for several promising future enhancements that would elevate it to a production-ready, enterprise-grade security solution.

7.3.1 Hardware Security Module (HSM) Integration

The most critical future enhancement is to abstract the private key operations and integrate support for Hardware Security Modules. An HSM is a physical device that safeguards and manages digital keys and performs cryptographic operations. By integrating HSM support (e.g., via the PKCS#11 standard), the private key would never be exposed in software, system memory, or on disk. All signing operations would be offloaded to the tamper-resistant hardware, providing the highest possible level of key security and addressing the primary limitation of the current system.

7.3.2 Support for Additional Curves (e.g., Curve25519)

The framework could be extended to support other modern elliptic curves, particularly Curve25519 as used in the Edwards-curve Digital Signature Algorithm (EdDSA). EdDSA offers several advantages over ECDSA, including higher performance and intrinsic resistance to certain classes of implementation errors, such as the danger of a faulty random number generator for nonces. Expanding the framework to support EdDSA would provide users with more flexibility and an even stronger security posture.

7.3.3 Formal Verification of Log Integrity

For the secure logging suite, a dedicated verification tool could be developed to perform a formal audit of an entire chain of log files. This tool would go beyond verifying individual signatures. It would process a sequence of log chunks (e.g., `chunk_0001.log`, `chunk_0002.log`, etc.) and verify not only every signature within them but also the integrity of the sequence itself by checking for contiguous, non-repeating timestamps or explicit sequence numbers. This would create a complete, provable audit trail, making it possible to mathematically prove that no log entries were deleted, inserted, or reordered after the fact.

References

[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.