Job Blockchain System
Technical Documentation

1. Overview of Blockchain Principles Implementation

The Job Blockchain System implements core blockchain principles to create a tamper-evident job listing database. Here's how blockchain fundamentals are applied in this application:

1.1. Block Structure

Each job listing represents a "block" in the blockchain, containing:

- Job details (ID, title, company, location, description)
- Timestamp of creation
- Hash of the previous block (`prevHash`)
- Hash of the current block (`currentHash`)

1.2. Chain Formation

The blockchain is implemented as a linked list where:

- Each new job listing is added to the end of the chain
- The first block (genesis block) has a predefined `prevHash` of all zeros
- Each subsequent block references the hash of the preceding block

1.3. Immutability

The system ensures data immutability through:

- Cryptographic hashing of block contents
- Linking each block to the previous one via hash references
- Integrity verification that detects any modifications to existing blocks

2. Cryptographic Hashing Mechanism

2.1. Hash Implementation

The system uses SHA-256 hashing (via OpenSSL's EVP interface) to create unique fingerprints of each block:

```c
void generate_hash(const char *input, char *output)
{
        EVP_MD_CTX *mdctx;
        const EVP_MD *md;
        unsigned char hash[EVP_MAX_MD_SIZE];
        unsigned int hash_len;
        unsigned int i;

        mdctx = EVP_MD_CTX_new();
        md = EVP_sha256();

        EVP_DigestInit_ex(mdctx, md, NULL);
        EVP_DigestUpdate(mdctx, input, strlen(input));
        EVP_DigestFinal_ex(mdctx, hash, &hash_len);

        for (i = 0; i < hash_len; i++)
                sprintf(output + (i * 2), "%02x", hash[i]);
        output[64] = '\0';

        EVP_MD_CTX_free(mdctx);
}
```

2.2. Block Hashing Process

Each block's hash is calculated from multiple data points:

```c
void calculate_hash(JobListing *job)
{
        char buffer[MAX_TITLE_LENGTH + MAX_COMPANY_LENGTH +
                    MAX_LOCATION_LENGTH + MAX_DESCRIPTION_LENGTH +
                    HASH_LENGTH + 100];

        sprintf(buffer, "%d%s%s%s%s%ld%s", job->id, job->title,
                job->company, job->location, job->description,
                job->timestamp, job->prevHash);

        generate_hash(buffer, job->currentHash);
}
```

This comprehensive approach ensures that any change to any field would result in a completely different hash value.

2.3. Importance of Data Integrity

The cryptographic hashing mechanism is crucial for maintaining data integrity in several ways:

1. **Tamper Detection**: Any modification to a block's data will change its hash, making tampering evident
2. **Chain Validation**: The prevHash-currentHash relationship verifies the entire chain's integrity
3. **Data Authenticity**: The hash serves as a digital fingerprint that confirms data hasn't been altered
4. **Consistency Verification**: The integrity check can recalculate hashes to verify they match stored values

3. Integrity Verification Process

The system implements a robust two-pass integrity verification process:

3.1. **First Pass**: Internal Block Integrity

```
strcpy(temp_hash, current->currentHash);
calculate_hash(current);
if (strcmp(temp_hash, current->currentHash) != 0)
{
        fprintf(output_file, "  HASH TAMPERING: Recalculated hash doesn't match stored hash\n");
        fprintf(output_file, "  Stored hash: %s\n", temp_hash);
        fprintf(output_file, "  Recalculated hash: %s\n", current->currentHash);
        is_valid = 0;
}
else
{
        /* Restore the original hash */
        strcpy(current->currentHash, temp_hash);
}

current = current->next;
```

This verifies that each block's stored hash matches its recalculated hash, detecting any tampering with block content.

3.2. **Second Pass**: Chain Integrity

```
/* Second pass: check chain integrity - this matches original logic */
current = chain->head;
while (current->next != NULL)
{
        /* Check if the prevHash of the next block matches this block's currentHash */
        if (strcmp(current->currentHash, current->next->prevHash) != 0)
        {
                fprintf(output_file, "\nBlock #%d -> Block #%d integrity error:\n",
                        block_count - (current->next->next ? 2 : 1),
                        block_count - (current->next->next ? 1 : 0));
                fprintf(output_file, "  Current block hash: %s\n", current->currentHash);
                fprintf(output_file, "  Next block prevHash: %s\n", current->next->prevHash);
                fprintf(output_file, "  HASH MISMATCH: Chain integrity compromised\n");
                is_valid = 0;
        }

        current = current->next;
}
```

This ensures that each block correctly references the previous block, maintaining the chain's integrity.

4. Challenges Faced and Solutions

4.1. Block Order and Chain Direction

Challenge: The initial implementation had a critical flaw: New blocks were added to the front of the chain (head), creating a reversed blockchain structure that didn't align with the verification logic.

Solution: Modified the `add_job_listing` function to:

- Add new blocks to the end of the chain
- Traverse to find the last block
- Set the new block's `prevHash` to the last block's `currentHash`

- Link the last block to the new block

4.2. Hash Integrity During Verification

Challenge: Recalculating a block's hash during verification would overwrite the original hash value, potentially hiding tampering evidence in subsequent verification steps.

Solution: Implemented a hash preservation mechanism:

- Save the original hash before recalculation
- Compare with recalculated hash to detect tampering
- Restore the original hash value after comparison

This ensures accurate verification without side effects.

4.3. Detailed Diagnostic Information

Challenge: Initial verification only reported binary success/failure without details on where or how the chain was compromised.

Solution: Enhanced the verification process to:

- Output detailed block information
- Identify specific integrity issues (hash tampering vs. chain breaks)
- Provide the precise location of integrity violations
- Show both the expected and actual hash values

The results are now written to an output file (`output.txt`) with clear diagnostic information.

The Job Blockchain System demonstrates the fundamental principles of blockchain technology in a practical application. By implementing cryptographic hashing, linked block structure, and integrity verification, the system provides a tamper-evident record of job listings that ensures data integrity and authenticity.