Hash Chain and Digital Signature Scheme for Tamper-Evident Product Provenance Tracking on Permissioned Blockchain

Leonard Arif Sutiono - 18223120
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 18223120@std.stei.itb.ac.id, leouwse@gmail.com

***Abstract*—Food supply chains are vulnerable to silent data falsification: a courier or warehouse operator who mishandles a perishable shipment, for instance by failing to keep a refrigerated truck at the required temperature, has both the opportunity and the incentive to alter the recorded condition data after the fact to avoid liability. This paper presents the design and implementation of a blockchain-based integrity layer for a web-based food supply chain tracking system, intended to make such retroactive falsification cryptographically detectable. Every checkpoint event, including courier check-ins recording temperature, humidity, and location, is hashed using SHA-256, linked to the previous block through a hash chain, and signed using the Elliptic Curve Digital Signature Algorithm (ECDSA) over the P-256 curve to bind the record to the courier who submitted it. The system exposes a Ledger Explorer interface that renders each block as a card showing its index, timestamp, transaction data, previous hash, current hash, and signature, alongside a deliberately provided "Illegal Edit" function that lets a user simulate a tampering attack by modifying a block's stored data without recalculating its hash. A recursive validation routine then detects the resulting hash mismatch and visually marks the tampered block and every subsequent block as corrupted. Implementation was carried out in Go, and the cryptographic primitives were evaluated for correctness and tamper-detection reliability. The result demonstrates that a lightweight, locally implemented hash chain combined with digital signatures is sufficient to guarantee the immutability and authenticity of food logistics records without requiring a public blockchain network.**

***Keywords*—Blockchain; SHA-256; Hash Chain; ECDSA; Digital Signature; Data Integrity; Tamper Detection; Food Supply Chain**

I. Introduction

Perishable food logistics depends on accurate condition records: the temperature, humidity, and timing of every checkpoint a shipment passes through determine whether the product is still safe to sell upon arrival. The party who created these records, however, is frequently the same party who would be blamed if the records revealed a problem. A courier who allowed a refrigerated truck to warm above the safe threshold has a direct incentive to quietly revise the logged temperature after the fact, rewriting history so that the spoiled shipment appears to have been handled correctly the entire time. In a conventional database-backed tracking system, such a revision is trivial: an UPDATE statement on a temperature column leaves no trace that anything was ever different.

This paper addresses that problem by designing and implementing a blockchain-style integrity layer purpose-built for a food supply chain tracking system. Rather than deploying a public blockchain network, which is unnecessary overhead for a system whose participants, factories, warehouses, and retailers in a defined logistics network, are already known to one another, the system implements a local hash chain combined with per-record digital signatures. Each checkpoint event, captured through a courier check-in form recording temperature, humidity, location, and timestamp, is bundled into a block. Every block stores a SHA-256 hash of its own contents together with the hash of the block immediately preceding it, forming an unbroken chain in which altering any single block's data changes that block's hash and breaks the link to every block that follows.

Hashing alone proves that a block's stored data matches its stored hash, but it does not prove who submitted that data in the first place. To address authenticity, every block is additionally signed using the Elliptic Curve Digital Signature Algorithm (ECDSA) over the P-256 curve, binding the record to a specific courier's private key. A verifier holding the corresponding public key can confirm that the data was indeed submitted by that courier and has not been altered since signing.

The contribution of this paper is threefold. First, a concrete hash-chain block structure and validation routine are designed and implemented specifically for perishable food checkpoint data, distinct from a general-purpose cryptocurrency ledger. Second, an ECDSA-based signing and verification scheme is integrated so that every checkpoint record carries proof of its courier's identity in addition to its content integrity. Third, a tamper-detection experiment is conducted in which a block's data is deliberately altered without recalculating its hash, demonstrating that the resulting hash mismatch is reliably detected and visually surfaced to the user through a web-based Ledger Explorer interface.

The remainder of this paper is organized as follows. Section II reviews the theoretical foundations of cryptographic hash functions, hash chains, and ECDSA. Section III describes the system design and implementation. Section IV presents the experimental setup, results, and analysis of the tamper-detection mechanism. Section V concludes the paper and outlines directions for future work.

II. Theoretical Foundations

A. Cryptographic Hash Functions and SHA-256

A cryptographic hash function maps an input of arbitrary length to a fixed-length output, called a digest, such that the mapping is deterministic, computationally infeasible to invert, and highly sensitive to any change in the input, a property known as the avalanche effect: flipping even a single bit of the input produces a digest that differs unpredictably across roughly half of its output bits [1]. The Secure Hash Algorithm 256-bit (SHA-256), part of the SHA-2 family standardized by NIST, produces a 256-bit digest through a compression function applied over sixty-four rounds of bitwise operations, modular additions, and logical functions on 32-bit words [2]. SHA-256 is the digest function used throughout this system, both to fingerprint individual food items and to compute the per-block hash that forms the backbone of the chain.

B. Hash Chains and Blockchain Structure

A hash chain is a sequence of data blocks in which each block stores the hash of the block immediately before it, in addition to its own data and its own hash. This construction, popularized by blockchain systems such as Bitcoin, creates a dependency in which the hash of block i is computed over the concatenation of block i's own fields and the hash of block $i-1$ [3]. Consequently, modifying the stored data of any block changes that block's hash, which in turn no longer matches the previous-hash field stored in the next block, causing every subsequent block in the chain to become detectably inconsistent. This cascading effect is precisely what gives a hash chain its tamper-evident property: an attacker cannot silently rewrite history at one point in the chain without also being forced to recompute every hash from that point forward, an operation the legitimate validator can immediately catch by recomputing and comparing hashes.

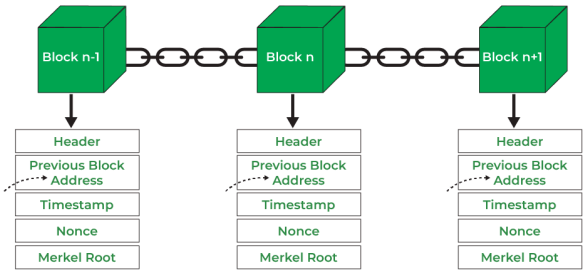


Fig. 1. Block and hash chain structure used in the system, linking each block's stored hash to the previous block.
(Source: GeeksforGeeks)

In this system, a block does not represent a financial transaction as in a cryptocurrency ledger, but rather a single supply chain event: a courier check-in recording the temperature, humidity, location, and timestamp of a food item at a specific point in its journey. The chain as a whole therefore forms an append-only, verifiable audit trail of every checkpoint a shipment has passed through.

C. Elliptic Curve Digital Signature Algorithm (ECDSA)

A digital signature scheme allows a party holding a private key to produce a signature over a message such that anyone holding the corresponding public key can verify that the message was signed by that private key and has not been altered, without ever learning the private key itself [4]. The Elliptic Curve Digital Signature Algorithm achieves this using the algebraic structure of an elliptic curve over a finite field rather than the integer factorization or discrete logarithm problems used by RSA or classical Diffie-Hellman, allowing ECDSA to achieve a comparable security level with substantially shorter keys [5].

This system uses curve P-256 (also known as secp256r1), a NIST-standardized curve offering 128-bit security strength. To sign a message m, the signer computes a hash digest

$$h = SHA256(m)$$

then computes a signature pair (r, s) using their private key d and a randomly chosen nonce k, according to the standard ECDSA signing equations over the curve's base point G and order n. Verification recomputes a value from the public key, the signature pair, and the same digest h, and accepts the signature only if that value matches r.

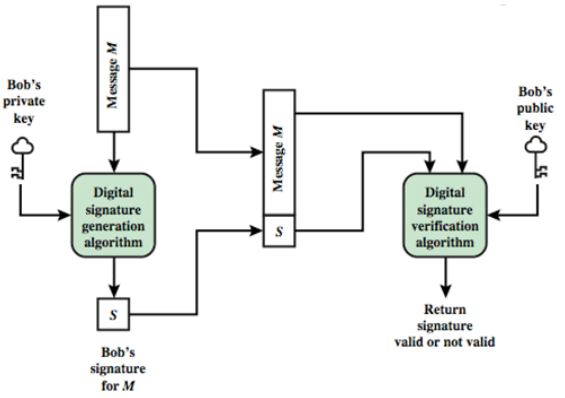


Fig. 2. ECDSA signing and verification flow: a private key signs a hash digest, and a public key verifies it.
(Source: ResearchGate)

In this system, every block's hash is treated as the message to be signed: after a block's SHA-256 hash is computed, that hash digest is itself signed with the submitting courier's ECDSA private key, and the resulting signature is stored alongside the block. This couples content integrity, provided by the hash chain, with authenticity, provided by the signature, so that a verifier can confirm both that a block's data has not changed and that it was genuinely submitted by the courier it claims to be from.

III. System Design and Implementation

A. System Overview

The cryptographic integrity layer is one component of a larger web-based food supply chain tracking system. The backend is implemented in Go and is organized into three cooperating modules: hash.go for SHA-256 hashing utilities, ecdsa.go for ECDSA key generation, signing, and verification, and blockchain.go for block creation, chain

validation, and tamper simulation. The frontend, built with React and TypeScript, exposes a Ledger Explorer interface that renders the chain visually and provides the tamper-detection demonstration described in Section IV. The route-optimization layer of the system, which determines how shipments are routed between distribution nodes, is discussed in a separate companion paper.

B. Block Structure and Hashing

Each block stores an Index identifying its position in the chain, a Timestamp recorded at creation time, a TransactionData payload, a PrevHash field copied from the previous block's hash, a Hash field, a Signature field, and a Valid flag used by the frontend to render the block's status. The TransactionData payload carries the fields relevant to a food checkpoint event: food identifier, location, temperature, humidity, expiry date, courier identifier, and event type.

The block hash is computed by the HashBlock function, which serializes the block's index, timestamp, JSON-encoded transaction data, and previous hash into a single string and passes it through SHA-256 via the shared CalculateHash function. A separate HashFood function fingerprints a food item independently of any block, by hashing the concatenation of its identifier, name, and expiry date, which is used elsewhere in the system to uniquely identify a product regardless of how many checkpoint blocks reference it.

```
func CalculateHash(data string) string {
  dataByte := []byte(data)
  hashData := sha256.Sum256(dataByte)
  return hex.EncodeToString(hashData[:])
}

func HashBlock(block models.Block) string {
  payload := fmt.Sprintf("%d|%s|%s|%s",
block.Index, block.Timestamp,
mustJSON(block.Data), block.PrevHash)
  return CalculateHash(payload)
}

func HashFood(food models.Food) string {

  expiryStr :=
food.ExpiryDate.Format(time.RFC3339)

  return
CalculateHash(fmt.Sprintf("%s|%s|%s",food.ID,
food.Name, expiryStr))
}

func mustJSON(value any) string {
  bytes, err := json.Marshal(value)
  if err != nil {
    panic(err)
  }
  return string(bytes)
}
```

Fig. 3. SHA-256 hashing implementation, including CalculateHash, HashBlock, and HashFood.
(Source: User's Code)

C. Key Generation, Signing, and Verification

The ecdsa.go module generates a P-256 key pair for each courier using Go's standard crypto/ecdsa and crypto/elliptic packages. The private key is serialized with x509.MarshalECPrivateKey and the public key with x509.MarshalPKIXPublicKey, both encoded as hexadecimal strings so they can be stored and transmitted as plain text alongside the rest of the system's data.

To sign a block, the SignData function first computes CalculateHash(data) to obtain a SHA-256 digest of the input as a hexadecimal string, then decodes that hex string back into raw bytes before passing it to ecdsa.Sign together with the courier's private key. The resulting (r, s) pair is encoded as a single string by converting each component to its hexadecimal representation and joining them with a period delimiter. VerifySignature reverses this process: it splits the signature string back into r and s, decodes the provided public key, recomputes the same hash digest over the input data, and calls ecdsa.Verify to confirm the signature.

```
func GenerateKeyPair() (privateKey
string, publicKey string) {
  privKey, err :=
ecdsa.GenerateKey(elliptic.P256(),
rand.Reader)
  if err != nil{
    return "",""
  }

  privBytes, err :=
x509.MarshalECPrivateKey(privKey)
  if err != nil{
    return "",""
  }

  privateKeyHex :=
hex.EncodeToString(privBytes)

  pubBytes, err :=
x509.MarshalPKIXPublicKey(&privKey.Public
Key)
  if err != nil{
    return "",""
  }

  publicKeyHex :=
hex.EncodeToString(pubBytes)

  return privateKeyHex, publicKeyHex
}

func SignData(data, privateKey string)
string {
  privBytes, err :=
hex.DecodeString(privateKey)
  if err != nil{
    return ""
  }

  privKey, err :=
x509.ParseECPrivateKey(privBytes)
  if err != nil{
    return ""
  }

  hashHex := CalculateHash(data)
  hashedData, err :=
hex.DecodeString(hashHex)
  if err != nil{
    return ""
  }
```

```
  r, s, err :=
ecdsa.Sign(rand.Reader,privKey,
hashedData[:])
  if err != nil{
    return ""
  }

  signature := r.Text(16) + "." +
s.Text(16)

  return signature
}

func VerifySignature(data, signature,
publicKey string) bool {
  parts := strings.Split(signature, ".")

  if len(parts) != 2 {
    return false
  }

  r, check :=
new(big.Int).SetString(parts[0], 16)
  if !check {
    return false
  }

  s, check :=
new(big.Int).SetString(parts[1], 16)
  if !check{
    return false
  }
  pubBytes, err :=
hex.DecodeString(publicKey)
  if err != nil{
    return false
  }
  pubParsed, err :=
x509.ParsePKIXPublicKey(pubBytes)
  if err != nil{
    return false
  }

  pubKey, check :=
pubParsed.(*ecdsa.PublicKey)
  if !check{
    return false
  }

  hashHex := CalculateHash(data)
  hashedData, err :=
hex.DecodeString(hashHex)
  if err != nil{
    return false
  }


  return
ecdsa.Verify(pubKey,hashedData[:],r,s)
}
```

Fig. 4. ECDSA key generation, signing, and verification implementation.
(Source: User's Code)

One implementation detail worth noting is that the message actually fed into ecdsa.Sign and ecdsa.Verify is not the raw input string but the byte representation of its SHA-256 hex digest, since CalculateHash is applied before the elliptic-curve signing step. This means the system effectively signs over the decoded bytes of a hex-encoded hash rather than the raw 32-byte digest directly, a deliberate layering of the shared hashing utility underneath the signing routine so that both the chain's hash field and the signed message are derived consistently from the same CalculateHash function.

D. Blockchain Management and Validation

The blockchain.go module exposes three operations. AddBlock constructs a new block from incoming transaction data, copies the hash of the current last block into the new block's PrevHash field (or leaves it empty for the genesis block), computes the new block's hash with HashBlock, signs that hash with the courier's private key, and appends the result to the chain.

ValidateChain iterates over every block in the chain and performs two checks per block: it recomputes the block's hash with HashBlock and compares it against the stored Hash field, and it confirms that the block's PrevHash field matches the actual hash of the preceding block (or is empty, for the first block). The function returns as soon as it finds the first block that fails either check, reporting that block's index as the point of corruption.

TamperBlock exists specifically to simulate an attack: it overwrites a target block's TransactionData with new, falsified values without recalculating that block's Hash field, exactly modeling a courier who edits a database record directly rather than going through the proper signing workflow. After the overwrite, the function re-runs ValidateChain and marks every block from the point of corruption onward as invalid.

```
func AddBlock(chain []models.Block, data
models.TransactionData, privateKey
string) models.Block {
  index := len(chain)
  prevHash := ""
  if len(chain) > 0 {
    prevHash = chain[len(chain)-1].Hash
  }

  block := models.Block{
    Index:     index,
    Timestamp:
time.Now().UTC().Format(time.RFC3339),
    Data:      data,
    PrevHash:  prevHash,
    Valid:     true,
  }
  block.Hash = HashBlock(block)
  block.Signature = SignData(block.Hash,
privateKey)
  return block
}

func ValidateChain(chain []models.Block)
(bool, int) {
  for i, block := range chain {
    expectedHash := HashBlock(block)
    if block.Hash != expectedHash {
      return false, i
    }
```

```
    if i == 0 {
      if block.PrevHash != "" {
        return false, i
      }
      continue
    }
    if block.PrevHash != chain[i-1].Hash
{
      return false, i
    }
  }
  return true, -1
}

func TamperBlock(chain []models.Block,
index int, newData
models.TransactionData) []models.Block {
  if index < 0 || index >= len(chain) {
    return chain
  }

  chain[index].Data = newData
  valid, invalidIndex :=
ValidateChain(chain)
  for i := range chain {
    chain[i].Valid = valid || i <
invalidIndex
  }
  return chain
}
```

Fig. 5. Blockchain management implementation: AddBlock, ValidateChain, and TamperBlock.
(Source: User's Code)

E. Courier Check-In and Ledger Explorer Interface

On the frontend, a courier check-in form captures the food item identifier, current location, actual temperature, humidity, and an automatically generated timestamp. Submitting the form calls the backend's check-in endpoint, which invokes AddBlock and appends the resulting block to the chain.

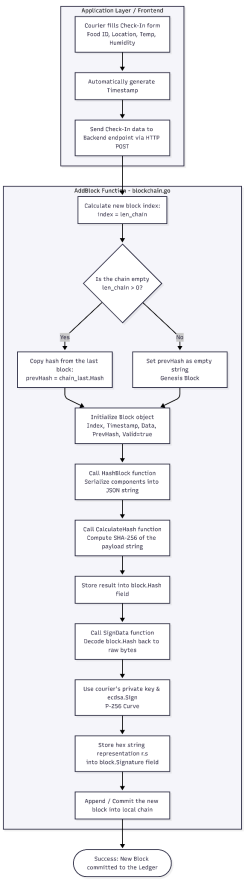


Fig. 6. Flow from a courier check-in submission to the resulting signed block being appended to the chain.

The Ledger Explorer module renders every block in the chain as an individual card displaying its index, timestamp, transaction data, previous hash, current hash, and signature. A dedicated "Illegal Edit" control is attached to each card, allowing a user to invoke the TamperBlock operation directly from the interface as a deliberate, self-contained demonstration of the system's tamper-detection capability, described further in Section IV.

Fig. 7. Web-based Ledger Explorer interface showing block cards with hash and signature fields.

IV. Experimental Results and Analysis

A. Experimental Setup

Two experiments were conducted. The first is a functional tamper-detection test: a chain of valid blocks is constructed through normal courier check-ins, after which a single block partway through the chain is deliberately tampered using the "Illegal Edit" function, simulating the motivating scenario from Section I in which a courier rewrites a recorded temperature to avoid liability for a spoiled shipment. The chain is then re-validated to confirm that the tampered block and every block after it are correctly flagged as invalid. The second experiment is

a performance benchmark measuring the time required to sign and verify blocks as the chain grows, to characterize the computational overhead introduced by the cryptographic layer.

B. Tamper Detection Results

Prior to tampering, ValidateChain was expected to report the entire chain as valid, with every block's recomputed hash matching its stored hash and every PrevHash field correctly linking to its predecessor. After invoking the "Illegal Edit" function on an intermediate block, for instance changing a recorded temperature of 4°C to 3°C, the block's stored Hash field no longer matches a freshly recomputed HashBlock over its new (falsified) data, and this mismatch was expected to be detected as soon as ValidateChain reaches that block.

Fig. 8. Chain state before tampering: all blocks display a VALID status.

Fig. 9. Chain state after the illegal edit: the tampered block and all subsequent blocks turn red and display a corrupted status.

Because every block after the tampered one stores the original (now incorrect) PrevHash inherited from the chain's construction at that time, the cascading failure described in Section II-B was expected to propagate forward, causing the interface to mark not just the directly edited block but every block chained after it as corrupted, consistent with the design goal that an attacker cannot localize the damage of a single falsified record.

C. Discussion

The tamper-detection experiment directly demonstrates the system's core security property: a hash chain does not prevent an attacker from editing stored data, since the TamperBlock function shows that doing so is trivially possible at the storage layer, but it does guarantee that any such edit is mathematically detectable the moment the chain is re-validated, because SHA-256's avalanche property ensures that even a one-degree change in a recorded temperature value produces a completely different hash output, which can no longer match the value frozen into the next block's PrevHash field.

The addition of ECDSA signatures contributes a property that hashing alone cannot provide: accountability. A hash mismatch proves that data has changed since the block was created, but it does not by itself prove who created the original, legitimate version. By requiring every block to be signed with the submitting courier's private key, the system ensures that a legitimate block can always be traced back to a specific, verifiable identity, which matters operationally since the goal is not merely to detect that food-safety data was falsified, but to be able to identify which courier's checkpoint record was tampered with.

V. Conclusion

This paper presented the design and implementation of a blockchain-style data integrity layer for a food supply chain tracking system, combining a SHA-256 hash chain with ECDSA digital signatures over curve P-256. Every courier check-in event recording temperature, humidity, and location is bundled into a block whose hash is linked to its predecessor, and every block is signed to bind it to the courier who submitted it. A tamper-simulation feature was implemented to allow direct demonstration of the chain's tamper-evident property: editing a block's stored data without recomputing its hash produces a detectable mismatch that the system's validation routine surfaces immediately, cascading visibly to every subsequent block in the chain. This shows that a lightweight, locally implemented hash chain is sufficient to guarantee the immutability and authenticity of perishable food checkpoint records without the overhead of a public, decentralized blockchain network, which is unnecessary in a closed logistics network where all participating parties are already known.

Future work includes extending the validation routine to recompute and reject signatures from revoked or rotated courier keys, integrating a Merkle tree structure to allow efficient partial verification of large chains without rehashing every block, and conducting a broader security evaluation against more sophisticated attacks such as an adversary who controls both the tampering and the re-signing step using a stolen private key.

VI. Appendix

Source code repository: https://github.com/LeonArif/SupplyChain

ACKNOWLEDGMENT

The author would like to express gratitude to the lecturers of II4021 Kriptografi for their guidance throughout the course and for providing the lecture materials referenced in this work, as well as to family and peers for their continuous support during the preparation of this paper.

REFERENCES

[1] R. Munir, "Fungsi Hash Kriptografi," [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2025-2026/. [Accessed: 16-Jun-2026].

[2] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," FIPS PUB 180-4, Aug. 2015.

[3] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[4] R. Munir, "Tanda Tangan Digital," [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2025-2026/. [Accessed: 16-Jun-2026].

[5] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," International Journal of Information Security, vol. 1, no. 1, pp. 36–63, 2001.

[6] National Institute of Standards and Technology, "Digital Signature Standard (DSS)," FIPS PUB 186-5, Feb. 2023.

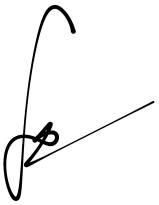
[7] GeeksforGeeks, "Blockchain Structure," [Online]. Available: https://www.geeksforgeeks.org/ethical-hacking/blockchain-structure/. [Accessed: 17-Jun-2026].

[8] ResearchGate, "Digital signature process - ECDSA has been established as an efficient algorithm against...," [Online]. Available: https://www.researchgate.net/figure/Digital-signature-process-ECDSA-has-been-established-as-an-efficient-algorithm-against_fig1_330978192. [Accessed: 17-Jun-2026].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 17 Juni 2026



Leonard Arif Sutiono 18223120