HKD_FACTOR_50: Secure Data Valuation and Sharing via Homomorphic Encryption

Michael S. Yang
Independent Researcher
yangofzeal@gmail.com

Abstract

Data marketplaces and collaborative analytics require a way to estimate the value of contributed records without exposing the records themselves. This paper presents HKD_FACTOR_50, a protocol architecture that combines task-specific marginal-contribution valuation, additively homomorphic aggregation, policy-bound release, and an auditable payment ledger. The construction does not claim that encryption makes arbitrary machine-learning training private. Instead, it protects a deliberately restricted interface: contributors encrypt quantized valuation statistics, an evaluator computes approved linear aggregates on ciphertexts, and a threshold or designated decryptor releases only authorized totals. We define correctness, confidentiality, integrity, and audit requirements; describe leakage and collusion boundaries; and provide a complete executable Python reference demonstration based on the Paillier cryptosystem. The program verifies encrypted weighted aggregation against plaintext computation and rejects tampered ledger entries. The reference code is pedagogical rather than production-hardened, but it makes the protocol's data flow and invariants reproducible. HKD_FACTOR_50 therefore supplies a concrete baseline for privacy-preserving data valuation while identifying the additional controls required for deployment: authenticated transport, robust key custody, range proofs, differential-privacy release, and independently reviewed cryptographic libraries.

Keywords—Data valuation; homomorphic encryption; privacy-preserving computation; Paillier cryptosystem; secure data sharing; auditability; data marketplace; Shapley value.

1 Introduction

Homomorphic encryption permits selected computations to be evaluated while inputs remain encrypted. Gentry's seminal construction established the feasibility of fully homomorphic encryption (FHE) for arbitrary circuits [1]. Subsequent standardization work has clarified security terminology, parameter selection, and deployment expectations [2]. This capability addresses only one part of secure data exchange: a useful marketplace must also define what "value" means, constrain the computation, authenticate participants, and control what is released.

Data valuation is inherently task-dependent. Data Shapley formalizes a datum's value as its average marginal contribution to a learning utility over coalitions [3]. Exact Shapley computation is exponential in the number of contributors, and approximation still requires careful treatment of the evaluation set and learning procedure. HKD_FACTOR_50 separates this valuation layer from its cryptographic transport layer. A valuation engine produces bounded, auditable sufficient statistics; the encryption layer then aggregates those statistics without disclosing each contributor's plaintext value.

The paper makes three contributions. First, it specifies a narrow protocol in which encrypted linear scoring is technically matched to an additive homomorphism. Second, it states explicit security and leakage boundaries instead of treating homomorphic encryption as a complete privacy solution. Third, it includes an executable reference implementation whose assertions test the central correctness and tamper-evidence properties.

2 Background and Design Basis

2.1 Valuation

Let $N = \{1, \ldots, n\}$ index contributed datasets and let $U(S)$ denote the utility of a model trained or evaluated with coalition $S \subseteq N$. The Shapley value of contributor i is

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(n - |S| - 1)!}{n!} [U(S \cup \{i\}) - U(S)]. \quad (1)$$

The axiomatic basis originates in cooperative game theory [4]. In practice, HKD_FACTOR_50 accepts a vector of bounded approximations $x_i = (x_{i1}, \ldots, x_{id})$, such as sampled marginal contributions, data-quality measures, or policy-approved counts. The protocol does not prescribe one universally correct utility function.

2.2 Homomorphic encryption

The executable construction uses Paillier encryption, whose ciphertext multiplication corresponds to plaintext addition and whose ciphertext exponentiation corresponds

to multiplication of a plaintext by a public integer [5]. For public key (n, g), plaintext $m \in \mathbb{Z}_n$, and random $r \in \mathbb{Z}^*_n$,

$$E(m; r) = g^m r^n \bmod n^2. \quad (2)$$

Thus $E(a)E(b) = E(a + b)$ and $E(a)^k = E(ka)$ in the plaintext ring. Approximate real-number workloads may instead use CKKS, which supports approximate arithmetic on packed values [6]; integer and exact modular workloads can use BFV-type leveled constructions [7]. Microsoft SEAL is one established implementation environment for BFV and CKKS [8]. Scheme substitution changes precision, circuit-depth, and parameter requirements, but not the separation between valuation policy and encrypted aggregation.

3 HKD_FACTOR_50 Protocol

3.1 Roles and workflow

The protocol defines four roles: contributors C_i, an evaluator E, a release authority R, and an append-only auditor A. A job identifier J binds every message to one valuation task.

1. R generates a key pair and publishes the public key, parameter profile, valuation specification, quantization scale q, bounds, and job identifier.
2. Each C_i computes or receives approved statistics x_i, verifies the bounds, encodes $m_{ij} = \lfloor q x_{ij} \rceil$, encrypts each coordinate, and signs the submission metadata.
3. E validates signatures and policy identifiers, then evaluates an approved linear score
$$V = \sum_{i=1}^{n} \sum_{j=1}^{d} w_j m_{ij} + b \quad (3)$$
by multiplying ciphertexts raised to public integer weights. Negative weights are represented modulo n.
4. R authorizes release, decrypts only the aggregate, converts signed modular output to an integer, divides by q, and records the result.
5. A verifies a hash-chained transcript containing commitments to policies, submissions, aggregate ciphertexts, releases, and payments.

The name "50" denotes a protocol profile identifier, not a claim of 50-bit security, 50-fold speed, or a new hardness assumption. Production security strength must follow current parameter guidance [2].

3.2 Quantization and range condition

For scale q, the decoded score differs from an ideal real-valued linear score by at most

$$\epsilon_q \le \frac{1}{2q} \sum_{i=1}^{n} \sum_{j=1}^{d} |w_j|, \quad (4)$$

assuming nearest-integer encoding. To avoid modular wraparound, the deployment must prove or conservatively enforce $|V| < n/2$. Production systems should attach zero-knowledge range proofs or use a trusted ingestion boundary; encryption alone does not prove that a contributor submitted an in-range value.

3.3 Payment rule

Let B be the buyer-funded pool, $c_i \ge 0$ the released contribution score, and $\tau \ge 0$ a minimum eligibility threshold. A transparent proportional rule is

$$p_i = \begin{cases} B \dfrac{\max(c_i - \tau, 0)}{D}, & D = \sum_k \max(c_k - \tau, 0) > 0, \\ 0, & \text{otherwise.} \end{cases} \quad (5)$$

Rounding, platform fees, dispute windows, and residual cents must be fixed in the signed policy. The rule is auditable but is not asserted to be uniquely fair.

4 Security Model

Confidentiality. Under Paillier's decisional composite residuosity assumption, a probabilistic polynomial-time evaluator without the private key cannot distinguish encryptions of chosen plaintexts beyond negligible advantage [5]. This protects submitted statistics in transit and during approved aggregation, subject to metadata and output leakage.

Correctness. For valid inputs satisfying the range condition, the decrypted aggregate equals the modular sum of encoded weighted inputs. The test program checks this invariant for positive and negative values.

Integrity and provenance. Homomorphic encryption is malleable by design and does not authenticate ciphertexts. HKD_FACTOR_50 therefore requires contributor signatures, replay-resistant job identifiers, authorization checks, and a hash-chained audit log. The demonstration implements only local hash-chain tamper evidence; it does not implement digital identity or remote attestation.

Collusion. A sole decryptor can decrypt individual submissions if it receives them. Strong deployments should use threshold decryption, organizational separation, and a policy engine that supplies only aggregate ciphertexts to the decryption quorum. Secure multiparty computation provides alternative trust distributions and can complement homomorphic encryption [9].

Output leakage. Repeated or differenced queries may reconstruct individual values. Query budgeting, minimum cohort sizes, correlated-query detection, and differential privacy are therefore required where released aggregates concern persons. The formal differential-privacy framework quantifies privacy loss under randomized release [10]; it is separate from ciphertext confidentiality.

5 Executable Reference Implementation

Listing 1 is verbatim Python 3 code using only the standard library. It generates a demonstration key, encrypts signed integers, computes a weighted encrypted sum, decrypts the result, verifies equality with plaintext computation, and detects a modified audit event. Set bits substantially higher and replace the program with an audited library for production. The small default makes the paper's test finish quickly and is intentionally not a production parameter.

Listing 1: Executable HKD_FACTOR_50 additive aggregation demonstration.

```
"""Educational HKD_FACTOR_50 demo; not production cryptography."""
from dataclasses import dataclass
from hashlib import sha256
from math import gcd
import json, secrets

def egcd(a, b):
    if b == 0:
        return a, 1, 0
    g, x, y = egcd(b, a % b)
    return g, y, x - (a // b) * y

def inv(a, n):
    g, x, _ = egcd(a % n, n)
    if g != 1:
        raise ValueError("inverse does not exist")
    return x % n

def is_probable_prime(n, rounds=32):
    if n < 2:
        return False
    small = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37)
    for p in small:
        if n == p:
            return True
        if n % p == 0:
            return False
    d, s = n - 1, 0
    while d % 2 == 0:
        s += 1
        d //= 2
    for _ in range(rounds):
        a = secrets.randbelow(n - 3) + 2
        x = pow(a, d, n)
        if x in (1, n - 1):
            continue
        for _ in range(s - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True

def prime(bits):
    while True:
        x = secrets.randbits(bits) | (1 << (bits - 1)) | 1
        if is_probable_prime(x):
            return x

@dataclass(frozen=True)
class PublicKey:
    n: int
    g: int

@dataclass(frozen=True)
class PrivateKey:
    lam: int
    mu: int

def keygen(bits=256):
    while True:
        p, q = prime(bits // 2), prime(bits // 2)
        if p != q and gcd(p * q, (p - 1) * (q - 1)) == 1:
            break
    n = p * q
    lam = (p - 1) * (q - 1) // gcd(p - 1, q - 1)
    g = n + 1
    nsq = n * n
    ell = (pow(g, lam, nsq) - 1) // n
    return PublicKey(n, g), PrivateKey(lam, inv(ell, n))

def encode_signed(x, n):
    if not (-n // 2 < x < n // 2):
        raise ValueError("signed plaintext is out of range")
    return x % n

def decode_signed(m, n):
    return m - n if m > n // 2 else m

def encrypt(pk, x):
    m, nsq = encode_signed(x, pk.n), pk.n * pk.n
    while True:
        r = secrets.randbelow(pk.n - 1) + 1
        if gcd(r, pk.n) == 1:
            return (pow(pk.g, m, nsq) * pow(r, pk.n, nsq)) % nsq

def decrypt(pk, sk, c):
    nsq = pk.n * pk.n
    ell = (pow(c, sk.lam, nsq) - 1) // pk.n
    return decode_signed((ell * sk.mu) % pk.n, pk.n)

def add(pk, *ciphertexts):
    nsq, out = pk.n * pk.n, 1
    for c in ciphertexts:
        out = (out * c) % nsq
    return out

def scale(pk, ciphertext, public_weight):
    nsq = pk.n * pk.n
    if public_weight < 0:
        ciphertext = inv(ciphertext, nsq)
        public_weight = -public_weight
    return pow(ciphertext, public_weight, nsq)

class AuditLog:
    def __init__(self):
        self.rows = []
    def append(self, event):
        previous = self.rows[-1]["hash"] if self.rows else "0" *
  64
        body = json.dumps(event, sort_keys=True, separators=(",",
 ":"))
        digest = sha256((previous + body).encode()).hexdigest()
        self.rows.append({"previous": previous, "event": event,
                          "hash": digest})
    def verify(self):
        previous = "0" * 64
        for row in self.rows:
            body = json.dumps(row["event"], sort_keys=True,
                              separators=(",", ":"))
            expected = sha256((previous + body).encode()).
  hexdigest()
            if row["previous"] != previous or row["hash"] !=
  expected:
                return False
            previous = row["hash"]
        return True

def main():
    pk, sk = keygen()
    values = [1250, -125, 875, 400]  # cents or scaled scores
    weights = [3, 2, -1, 4]
    encrypted = [encrypt(pk, x) for x in values]
    terms = [scale(pk, c, w) for c, w in zip(encrypted, weights)]
    encrypted_total = add(pk, *terms)
    observed = decrypt(pk, sk, encrypted_total)
    expected = sum(x * w for x, w in zip(values, weights))
    assert observed == expected

    log = AuditLog()
    log.append({"job": "demo-001", "event": "policy",
                "weights": weights})
    log.append({"job": "demo-001", "event": "release",
                "aggregate": observed})
    assert log.verify()
    log.rows[0]["event"]["weights"][0] = 99
    assert not log.verify()
    print({"expected": expected, "decrypted": observed,
           "tamper_detected": True})
```

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

6 Evaluation Method

The implementation establishes functional reproducibility, not comparative performance. A valid run prints identical plaintext and decrypted totals. For the included vector, the expected total is $3(1250) + 2(-125) - 875 + 4(400) = 4225$. The audit test then mutates the first recorded weight and requires verification to fail.

A production evaluation should report: key generation time; ciphertext size; encryption, aggregation, and decryption latency; peak memory; cohort size; feature dimension; quantization error; and end-to-end throughput. Measurements should identify hardware, software version, parameter set, warm-up policy, trial count, and confidence intervals. Security parameters must be selected with a recognized estimator and current standards rather than inferred from runtime alone.

7 Governance and Deployment Requirements

Encryption cannot establish data ownership, consent, legality, or semantic quality. Deployment therefore requires a signed data-use agreement, purpose limitation, retention and deletion rules, contributor withdrawal handling, jurisdiction-specific review, and a dispute process. The evaluator must publish the utility function, approximation procedure, random seeds or commitments, eligibility thresholds, and payment arithmetic before accepting submissions.

Operationally, the service should use authenticated encryption for network transport, hardware-backed or threshold key custody, role separation, dependency pinning, reproducible builds, incident response, and immutable external anchoring of audit checkpoints. Ciphertext validity and claimed input bounds require proofs or trusted validation. Differential privacy should be applied to releases when aggregate answers can expose individuals, with a documented privacy budget and composition policy [10].

8 Limitations

HKD_FACTOR_50, as specified here, evaluates linear functions over precomputed statistics. It does not privately train an arbitrary model, verify that statistics were honestly derived from raw records, hide public weights, or eliminate leakage from released outputs. Paillier lacks circuit privacy by default, and the pedagogical implementation omits constant-time arithmetic, secure memory handling, authenticated identities, threshold decryption, zero-knowledge proofs, and denial-of-service controls. Data Shapley approximations may be unstable under model, utility, and sampling choices. Finally, proportional payment can amplify noise and may be inappropriate where legal or contractual rights dominate marginal predictive utility.

9 Conclusion

HKD_FACTOR_50 defines a reproducible architecture for valuing and sharing approved data statistics without revealing each submitted value to the evaluator. Its central design choice is restraint: use additive homomorphic encryption for a bounded linear interface, make valuation and release policies explicit, and treat audit, authentication, output privacy, and governance as separate requirements. The included program demonstrates the algebraic core and tamper-evident transcript. A production implementation should replace the demonstration primitives with reviewed libraries and independently validate both cryptographic parameters and marketplace policy.

References

[1] C. Gentry, “Fully homomorphic encryption using ideal lattices,” in *Proc. 41st ACM Symp. Theory of Computing*, 2009, pp. 169–178.

[2] M. Albrecht et al., *Homomorphic Encryption Standard*, ver. 1.1, HomomorphicEncryption.org, 2024.

[3] A. Ghorbani and J. Zou, “Data Shapley: Equitable valuation of data for machine learning,” in *Proc. 36th Int. Conf. Machine Learning*, PMLR 97, 2019, pp. 2242–2251.

[4] L. S. Shapley, “A value for *n*-person games,” in *Contributions to the Theory of Games II*, H. W. Kuhn and A. W. Tucker, Eds. Princeton, NJ, USA: Princeton Univ. Press, 1953, pp. 307–317.

[5] P. Paillier, “Public-key cryptosystems based on composite degree residuosity classes,” in *Advances in Cryptology—EUROCRYPT ’99*, LNCS 1592, 1999, pp. 223–238.

[6] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers,” in *Advances in Cryptology—ASIACRYPT 2017*, LNCS 10624, 2017, pp. 409–437.

[7] J. Fan and F. Vercauteren, “Somewhat practical fully homomorphic encryption,” IACR Cryptology ePrint Archive, Rep. 2012/144, 2012.

[8] Microsoft Research, “Microsoft SEAL: Fast and easy-to-use homomorphic encryption library,” documentation, accessed Sep. 13, 2026.

[9] O. Goldreich, S. Micali, and A. Wigderson, “How to play any mental game,” in *Proc. 19th ACM Symp. Theory of Computing*, 1987, pp. 218–229.

[10] C. Dwork, F. McSherry, K. Nissim, and A. Smith, “Calibrating noise to sensitivity in private data analysis,” in *Theory of Cryptography Conference*, LNCS 3876, 2006, pp. 265–284.