



RESEARCH ARTICLE

OPEN ACCESS

EXPERIMENTAL EVALUATION OF SMART CONTRACT GAS OPTIMIZATION TECHNIQUES: A CASE STUDY OF A BLOCKCHAIN-BASED REPORTING APPLICATION SYSTEM

Michael Angello Qadosy Riyadi*¹, Sulthonika Mahfudz Al Mujahidin², Nur Alifia Rustan³
 A'isyah Belqis Febi Aulia⁴, Mastuty Ayu Ningtyas⁵ and Khodijah Amiroh⁶

^{1, 2, 3, 4, 5, 6}Department of Information Technology, Telkom University, Surabaya Campus, Surabaya, Indonesia.

¹<https://orcid.org/0009-0003-0853-0989> , ²<https://orcid.org/0009-0000-5873-1082> , ³<https://orcid.org/0009-0006-8714-5684> 
⁴<https://orcid.org/0009-0007-0786-1443> , ⁵<https://orcid.org/0009-0007-8501-5488> , ⁶<https://orcid.org/0000-0001-9498-0553> 

E-mail: *michaelangelo@student.telkomuniversity.ac.id, sulthonikamahfudzam@student.telkomuniversity.ac.id,
nuralifialrs@student.telkomuniversity.ac.id, aisyahbelqis@student.telkomuniversity.ac.id, mastutyayu@telkomuniversity.ac.id,
dijaamirah@telkomuniversity.ac.id

ARTICLE INFO

Article History

Received on: March 05, 2026
 Reviewed on: April 04, 2026
 Accepted on: May 01, 2026
 Published on: July 31, 2026

Keywords:

Blockchain,
 Ethereum,
 Smart Contract,
 Optimization,
 Gas Consumption.

ABSTRACT

Blockchain-based applications often face scalability challenges due to high gas costs associated with on-chain storage in the Ethereum network, particularly for systems requiring continuous audit logging. This study evaluates smart contract optimization strategies to reduce gas consumption in a blockchain-based mobile reporting system for sexual violence cases. An experimental approach was conducted by implementing four Solidity smart contract models—Baseline, Merkle Tree Aggregation, Storage Optimized (Packed Struct and Enum Encoding), and Conditional Commit—and testing them under scenarios of 50, 100, 500, and 1000 transactions using identical datasets. Gas consumption, transaction cost, latency, and execution time were analyzed using descriptive statistics and non-parametric tests. The results show that the baseline approach produced the highest gas consumption (770,899,033 gas for 1,000 reports), while Storage Optimization reduced it to 177,566,016 gas and Merkle Tree Aggregation to 20,492,732 gas. The Conditional Commit method achieved the lowest gas consumption at 4,308,404 gas with an execution time of 1.08 seconds compared to 143.72 seconds in the baseline. Statistical tests confirmed significant differences among methods (Kruskal–Wallis $H = 166.93$, $p < 0.01$; Mann–Whitney $p < 0.01$), demonstrating that reducing on-chain transactions substantially improves gas efficiency and scalability.



Copyright ©2026 by authors and Galileo Institute of Technology and Education of the Amazon (ITEGAM). This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).

I. INTRODUCTION

Blockchain technology has rapidly evolved as a digital infrastructure that enables transparent, distributed, and immutable data recording across various decentralized systems [1-5]. Among the numerous existing blockchain platforms, Ethereum has become one of the most widely utilized due to its capability to execute smart contracts through the Ethereum Virtual Machine (EVM), enabling developers to build decentralized applications (decentralized applications/DApps) with computational logic that can be publicly verified [6-8]. Within the Ethereum ecosystem, every computational operation performed by a smart contract requires an execution fee known as gas, which functions as a pricing mechanism for computational resources within the network [9-11]. This gas mechanism is designed to prevent the abuse of network resources, such as denial-of-service (DoS) attacks, while simultaneously ensuring that each computational operation is paid proportionally by the user who triggers it [12], [13]. Although the gas mechanism plays an essential role in maintaining the security and stability of the Ethereum network, gas costs often become a significant barrier in the development of data-intensive blockchain applications [14], [15].

In particular, operations involving permanent data storage on the blockchain incur substantially higher costs compared to ordinary computational operations [16], [17]. For instance, storage instructions such as SSTORE in Ethereum can consume up to 20,000 gas for the first non-zero value write, making it one of the most expensive operations in smart contract execution [18]. Ni *et al.* (2025) noted that large data volumes in blockchain applications introduce scalability challenges due to the high cost of on-chain storage [19]. This condition poses serious challenges for applications that require continuous data recording, as high transaction costs may limit the scalability and operational sustainability of blockchain-based systems [20]. Various studies have also demonstrated that the majority of gas consumption in Ethereum smart contracts originates from operations related to accessing and modifying permanent storage, thereby making optimization of data storage structures an important factor in improving transaction cost efficiency in modern blockchain applications [21], [22]. Several studies have attempted to identify factors that influence gas consumption in Ethereum smart contracts. Empirical studies indicate that certain code implementation patterns in the Solidity programming language can result in higher gas consumption due to the use of inefficient data structures or repeated execution of storage operations [23], [24]. Previous research has also identified various code smells and design patterns that lead to gas inefficiency during smart contract execution [25]. In response to these issues, several optimization techniques have been proposed, including the use of fixed-length data types, packing variables into a single storage slot through variable packing techniques, and the application of static analysis to detect inefficient code patterns prior to the deployment process [26-28]. The results of these approaches indicate that code-level optimization can significantly reduce gas consumption without altering the functional logic of the implemented smart contract.

In addition to optimization at the code implementation level, another approach widely explored in the literature is reducing the number of on-chain transactions through data aggregation mechanisms [29-31]. In this context, cryptographic structures such as Merkle Trees are commonly used to summarize a collection of data into a single root hash value that is subsequently stored on the blockchain [32], [33]. This approach enables data integrity to be preserved without requiring the entire dataset to be stored directly on the blockchain. Park *et al.* (2022) designed an event-driven resource offloading solution in which DApp users only pay a predetermined minimal gas fee while most computational tasks are executed on external servers with lower gas costs [34]. Through the use of Merkle proofs, the system can verify the existence of a particular data element within a Merkle Tree structure with logarithmic complexity, making it highly efficient for systems that require verification of large volumes of data. Data aggregation techniques such as transaction batching or threshold commit mechanisms have also been proposed to reduce the frequency of transactions that need to be recorded on the blockchain, thereby overall reducing gas consumption and improving the scalability of blockchain systems [35]. Nevertheless, several research gaps remain that require further attention. Most previous studies focus solely on optimizing individual smart contracts without conducting a comparative evaluation of multiple optimization strategies within a consistent experimental framework. Furthermore, many studies evaluate optimization only at the code level without considering integration with data aggregation techniques or off-chain computation simultaneously. Therefore, this study aims to conduct a comparative evaluation of various smart contract optimization techniques in reducing gas consumption on the Ethereum network, particularly in the context of a mobile-based sexual violence reporting application that utilizes blockchain as an audit trail mechanism. This research is expected to provide a more comprehensive understanding of efficient smart contract design strategies and offer practical recommendations for developers in designing more scalable and cost-efficient blockchain applications.

II. MATERIALS AND METHODS

This study employs an experimental approach to evaluate the effectiveness of algorithm optimization techniques in Solidity smart contracts within a blockchain-based mobile application system for reporting sexual violence, with the overall research workflow presented systematically in Figure 1. The research stages begin with a literature review, followed by a system architecture analysis to identify the characteristics of the application as well as the weaknesses of the baseline smart contract, which stores all report data entirely on-chain using the string data type, resulting in high gas consumption due to inefficient storage structures. Based on these findings, three optimization techniques were designed, namely Merkle Tree Aggregation, Storage Optimization through Packed Struct and Enum Encoding, and Conditional Commit. The implementation phase includes the development of four smart contracts, consisting of one baseline and three optimized versions using Solidity version 0.8.19 on the Ethereum (ETH) blockchain network, which are subsequently tested under scenarios of 50, 100, 500, and 1000 transactions with identical datasets to maintain the consistency and validity of the comparison.

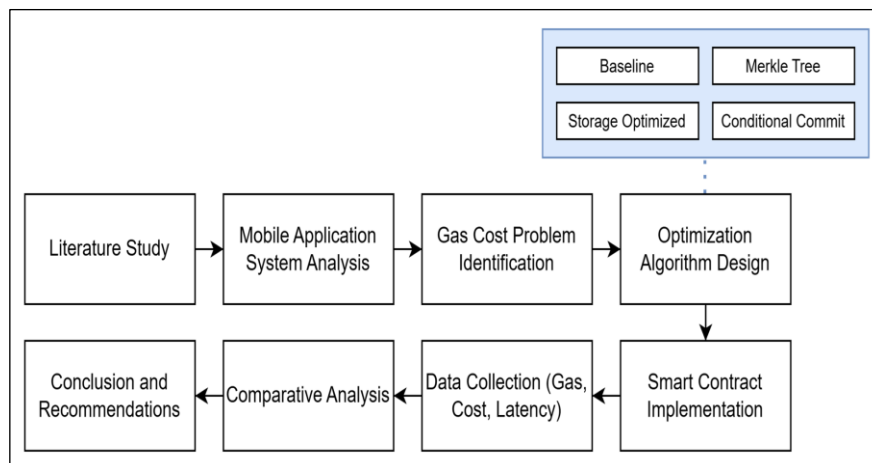


Figure 1: Research Flow.
Source: Authors, (2026).

II.1 SYSTEM OVERVIEW

The proposed system is a Flutter-based mobile application platform for reporting and monitoring cases of sexual violence within a university environment, utilizing the Ethereum blockchain to ensure data integrity and an immutable audit trail. Complete report data are stored off-chain using Google Firestore as a cloud-based NoSQL database, while integrity hashes and critical action logs are recorded on-chain through a smart contract to ensure transparency, security, and independent verification [36], [37], [38]. The system involves four main stakeholders with different roles and access rights as summarized in Table 1, while the main interface page of the application, which functions as the user access gateway, is shown in Figure 2.

Table 1: System Stakeholders and Users.

Stakeholder	Role	Feature Access
Reporter	Student or member of the academic community who reports an incident of violence, either as a direct victim or a witness.	Reporting form, status tracking
Admin	System manager responsible for receiving, processing, and following up on reports.	Admin panel, monitoring, report management, notifications
Psychologist	Professional personnel who provide psychological evaluation and assistance to victims.	Psychologist notes, consultation schedule
Blockchain System	Automated component that records every important action to the Ethereum smart contract as an immutable audit trail.	Hash recording, integrity verification

Source: Authors, (2026).

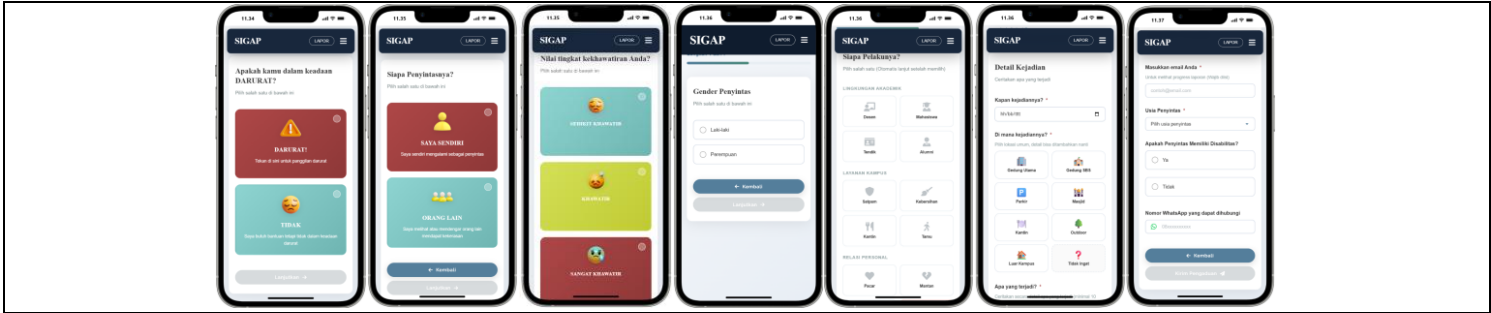


Figure 2: Main Interface of the Mobile Application for Reporting Sexual Violence.
Source: Authors, (2026).

II.2 SYSTEM ARCHITECTURE

The system is designed using a layered architecture approach consisting of three main layers. The first layer is the Presentation Layer based on Flutter with the Dart language, which is responsible for managing the user interface in the mobile application, including a multi-step reporting form, a report status tracking page, and a monitoring dashboard. The second layer is the Backend-as-a-Service Layer that utilizes Firebase, including authentication through Firebase Authentication, storage of complete report data off-chain in Cloud Firestore, as well as business logic and blockchain integration through Cloud Functions for Firebase based on Node.js to perform data validation, hash generation, and transaction submission to the Ethereum network. The third layer is the Blockchain Layer which functions to store integrity hashes, important metadata, and an immutable audit trail through smart contracts developed using Solidity, with all interactions conducted through a Web3 RPC provider [39], [40], while the overall structure and interaction flow between layers are comprehensively visualized in Figure 3.

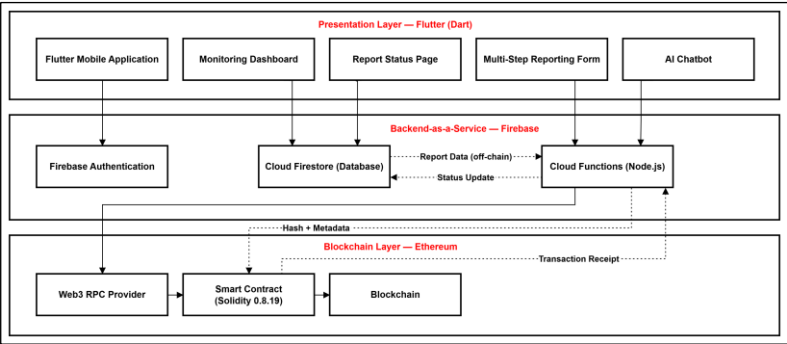


Figure 3: System Architecture Diagram.
Source: Authors, (2026).

II.3 REPORTING SYSTEM WORKFLOW

The reporting system workflow is designed to ensure data consistency, processing security, and blockchain-based recording integrity, and also serves as the basis for evaluating gas fee efficiency because each on-chain transaction directly contributes to gas consumption. The process begins with the completion of a multi-step form by the reporter in the mobile application, followed by advanced validation, input sanitization, generation of a unique report code, and storage of the data as the main document in the off-chain database.

Subsequently, the system generates a SHA-256 hash deterministically from the report data and sends it to the smart contract to be recorded as an audit log entry on the blockchain [41-43]. The transaction produces a receipt containing txHash, gasUsed, and blockNumber, which is then referenced back in the report document as an audit trail before the final response is sent to the application. The interaction flow among system components is shown in Figure 4, while the types of critical actions recorded on the blockchain and serving as the main sources of gas consumption are summarized in Table 2.

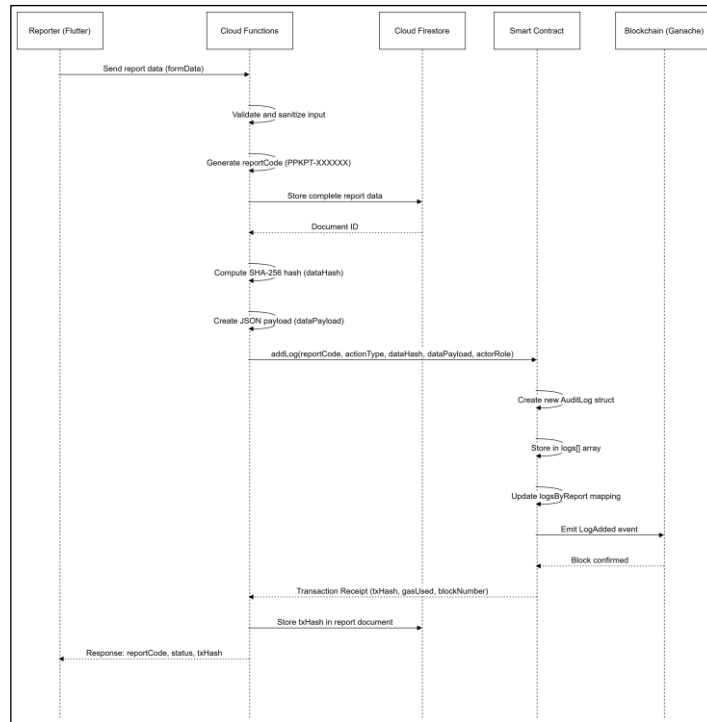


Figure 4: Sequence Diagram of Report Logging to Blockchain.
Source: Authors, (2026).

Table 2: Types of Actions Recorded on Blockchain.

Action Type	Trigger	Actor	Description
CREATE	User submits form	USER	New report created; data hash recorded on blockchain.
UPDATE_ADMIN	Admin updates status	ADMIN	Report status change recorded as audit trail.
NOTE_PSYCHOLOGIST	Psychologist writes note	PSYCHOLOGIST	Psychological evaluation note hashed and recorded.
FEEDBACK	Feedback submitted	USER/ADMIN	Feedback on case handling recorded.
DISPUTE	Objection submitted	USER	Reporter objection recorded as formal evidence.

Source: Authors, (2026).

II.4 SMART CONTRACT DESIGN

The baseline smart contract design in this study functions as the core mechanism for recording an immutable audit trail on the Ethereum network while simultaneously serving as the experimental control to identify sources of gas inefficiency and to compare the proposed optimization techniques. The contract is developed using Solidity with a storage scheme based on dynamic arrays and mappings to support chronological recording and per-report indexing [44], [45]. The primary data structure, *AuditLog*, represents a single activity entry containing the report identity, action type, integrity hash, complete data payload, timestamp, sender address, and actor role, as shown in Algorithm 1.

```

pragma solidity 0.8.19;
contract BaselineReporting {
    struct AuditLog {
        uint256 id; string reportCode; string actionType; string dataHash; string dataPayload; uint256 timestamp; address actor; string actorRole;
    }
    AuditLog[] public logs; mapping(string => uint256[]) public logsByReport;
    event LogAdded(uint256 indexed id, string reportCode, string actionType, address actor, uint256 timestamp);
}
  
```

Algorithm 1: Structure of *AuditLog* in *BaselineReporting* Smart Contract.
Source: Authors, (2026).

The *AuditLog* structure is designed by considering the storage characteristics of Ethereum, where fixed types such as *uint256* and *address* each occupy one 32-byte slot, whereas dynamic types such as *string* store the data length in the main slot and the actual content in separate storage slots. Each write or modification of a value in storage triggers an *SSTORE* (Storage Store) operation, which is an instruction that stores data into the permanent blockchain storage and represents one of the most gas-expensive operations.

Consequently, the use of several large string fields such as `dataPayload` and `dataHash` increases the number of slots utilized and enlarges gas consumption. The detailed storage characteristics of each field are presented in Table 3.

Table 3: Storage Characteristics of AuditLog Fields in Solidity.

Field	Solidity Type	Storage Characteristics	Description
id	uint256	Fixed 32 bytes (1 slot)	Sequential auto-generated ID
reportCode	string	Dynamic (≥ 32 bytes)	Report code, e.g., "PPKPT-100042" (13 bytes)
actionType	string	Dynamic (≥ 32 bytes)	Action type, e.g., "CREATE" (6 bytes)
dataHash	string	Dynamic (≥ 32 bytes)	SHA-256 hash in hexadecimal string (64 characters)
dataPayload	string	Dynamic (hundreds of bytes)	Complete JSON report data (~500–1000 characters)
timestamp	uint256	Fixed 32 bytes (1 slot)	Automatic block timestamp
actor	address	20 bytes (stored in 32-byte slot)	Ethereum address of transaction sender
actorRole	string	Dynamic (≥ 32 bytes)	Actor role, e.g., "ADMIN" (5 bytes)

Source: Authors, (2026).

Architecturally, the logs array is used to maintain chronological recording in an append-only manner, while the `logsByReport` mapping functions as a `reportCode`-based index to access the report history directly without full iteration. Each invocation of the `addLog()` function adds a new entry to both structures and emits an event as an external recording mechanism, so that within a single transaction several storage write operations occur that directly contribute to gas consumption. The implementation of this function is shown in Algorithm 2.

```
function addLog( string memory _reportCode, string memory _actionType, string memory _dataHash, string memory _dataPayload, string memory _actorRole)
public {
    uint256 newId = logs.length;
    AuditLog memory newLog = AuditLog({id: newId, reportCode: _reportCode, actionType: _actionType, dataHash: _dataHash, dataPayload: _dataPayload,
    timestamp: block.timestamp, actor: msg.sender, actorRole: _actorRole});
    logs.push(newLog); logsByReport[_reportCode].push(newId); emit LogAdded(newId, _reportCode, _actionType, msg.sender, block.timestamp); }
```

Algorithm 2: addLog Function Implementation in Smart Contract.

Source: Authors, (2026).

II.5 SMART CONTRACT ALGORITHM OPTIMIZATION METHODS

A comparative framework was designed to evaluate three optimization techniques against the baseline design by emphasizing differences in algorithmic characteristics that affect execution efficiency and the intensity of on-chain interactions. Four approaches were directly tested in a consistently controlled environment, so that the resulting performance variations fully reflect the data management strategies and transaction recording patterns implemented by each algorithm. Measurements were conducted automatically during the execution process and analyzed using descriptive statistics to obtain objective and measurable comparisons, while the structure of the experimental variables is summarized in Table 4.

Table 4: Experimental Variable Design.

Variable Type	Variable	Description
Independent	Smart contract algorithm type	4 levels: Baseline, Merkle Tree, Storage Optimized, Conditional Commit
Dependent	Gas Used	Total gas units consumed per transaction
	Gas Cost (ETH)	Transaction cost in ETH
	Latency	Transaction execution time in milliseconds
	Number of on-chain transactions	Total transactions submitted to blockchain
Control	Report data	Identical for all algorithms (deterministic input)
	Compiler	Solidity 0.8.19

Source: Authors, (2026).

II.5.1 Algorithm 1: Baseline — Individual Storage (Naive)

The baseline algorithm represents the simplest approach, in which each critical action generates a single on-chain transaction that directly records the entire report data individually. For each report, the system computes a SHA-256 hash as an integrity marker, constructs a complete JSON representation, and then sends all these parameters to the `addLog()` function in the smart contract. Because each report is processed and stored separately without aggregation or data structure compression, the number of transactions increases proportionally with the number of processed reports; therefore, this approach serves as the primary benchmark for measuring the efficiency of subsequent optimization techniques. The logical flow of the process is illustrated in Algorithm 3.

Input: A set of reports totaling n
Output: A list of transaction receipts
1. Retrieve one report from the dataset 2. Compute the SHA-256 hash of the report content as an integrity marker 3. Construct the report data in a complete JSON format 4. Send all information (report code, action type, hash, payload, actor role) to the smart contract 5. Wait until the transaction is confirmed on the blockchain and then store the transaction receipt 6. Repeat steps 1–6 until all n reports have been processed

Algorithm 3: Algorithm: BaselineReporting.

Source: Authors, (2026).

Computationally, the number of generated transactions is linear with respect to the number of reports, resulting in a complexity of $O(n)$, where n denotes the total number of processed reports. No aggregation or optimization mechanisms are implemented outside the contract; therefore, all interaction and storage costs are borne directly by individual transactions. The total gas consumption for processing n reports is formulated in Equation (1):

$$C_{baseline} = \sum_{i=1}^n (G_{base} + G_{calldata,i} + G_{storage,i}) \quad (1)$$

Which can be approximately simplified into Equation (2):

$$C_{baseline} \approx n \times C_{per_{tx}} \quad (2)$$

In this formulation, $C_{baseline}$ represents the total gas consumed for all reports. The component G_{base} represents the fixed base cost of transaction execution. The component $G_{calldata,i}$ denotes the cost resulting from transmitting the parameters of the i -th transaction, which is influenced by the size of the transmitted data. Meanwhile, $G_{storage,i}$ represents the permanent storage cost of the i -th transaction through the SSTORE operation, which has a high value because it involves writing data to blockchain storage. Furthermore, $C_{per_{tx}}$ represents the estimated average gas cost for a single individual transaction in the baseline approach.

II.5.2 Algorithm 2: Transaction Aggregation via Merkle Tree

This approach reduces the number of on-chain transactions by grouping several reports into a single batch, and then storing only one representative hash value in the form of a Merkle Root in the smart contract [46], [47]. With this strategy, the number of transactions decreases from n (as in the baseline) to $\lceil n/B \rceil$, where B denotes the batch size. Each batch is represented by the BatchCommit structure as shown in Algorithm 4, which stores the batch identity, the Merkle Root value, the number of reports, the range of report indices, the recording timestamp, and the address of the commitment sender. The aggregation procedure is illustrated in Algorithm 5. The Merkle Tree construction process is performed in a bottom-up manner, as illustrated in Algorithm 6.

```
struct BatchCommit { uint256 batchId; bytes32 merkleRoot; uint256 reportCount; uint256 fromIndex; uint256 toIndex; uint256 timestamp; address committer; }
```

Algorithm 4: BatchCommit Structure in MerkleAggregation Contract.

Source: Authors, (2026).

Input: n reports and batch size B
Output: List of batch transaction receipts
1. Determine the number of batches based on the total reports and the batch size 2. For each batch: a. Retrieve a group of reports according to its index range b. Convert each report into a hash (using KECCAK-256 over the report content) c. Arrange all hashes into a Merkle Tree structure d. Compute the root value of the tree (Merkle Root) e. Send the Merkle Root along with batch metadata to the smart contract through the commitBatch function f. Wait for transaction confirmation and store the transaction receipt 3. Return all batch transaction receipts

Algorithm 5: Algorithm: MerkleAggregation.

Source: Authors, (2026).

Input: List of report hashes in one batch
Output: A single Merkle Root value
1. Arrange all report hashes as the lowest level 2. Pair two adjacent hashes, then concatenate and hash them again to form a parent hash 3. If the number of elements is odd, duplicate the last hash to maintain pairing 4. Repeat the merging process until only one hash remains at the top 5. The final hash becomes the Merkle Root

Algorithm 6: Merkle Tree Construction Algorithm.

Source: Authors, (2026).

From a complexity perspective, constructing a Merkle Tree for a batch containing m reports requires $O(m \log m)$ because hashes are organized hierarchically from the leaf level to the root. With a fixed batch size B , the total off-chain computation for n reports becomes $O(n \log B)$. Meanwhile, on the on-chain side, the number of transactions decreases to $O(n/B)$, thereby reducing the frequency of global blockchain state writes proportionally to the batch size. The membership of a report within a batch is verified using a Merkle Proof as shown in Algorithm 7, by reconstructing the hash path from the leaf to the root using proof elements, and then comparing the final result with the Merkle Root stored in the contract. If both values are identical, the report is proven to be valid as part of the corresponding batch.

```

function verifyReport( bytes32[] memory _proof, bytes32 _leaf, uint256 _batchId)
public view returns (bool valid) {
    bytes32 computedRoot = _leaf;
    for (uint256 i = 0; i < _proof.length; i++) {
        bytes32 proofElement = _proof[i];
        if (computedRoot <= proofElement) { computedRoot = keccak256(abi.encodePacked(computedRoot, proofElement)); }
        else { computedRoot = keccak256(abi.encodePacked(proofElement, computedRoot)); }
    } return computedRoot == batches[_batchId].merkleRoot;
}

```

Algorithm 7: verifyReport Function for Merkle Proof Validation.

Source: Authors, (2026).

The total gas cost for the aggregation approach is formulated in Equation (3):

$$C_{\text{merkle}} = \left\lceil \frac{n}{B} \right\rceil \times C_{\text{store_root}} \quad (3)$$

Approximately, when n is much larger than B , the total cost can be expressed as Equation (4):

$$C_{\text{merkle}} \approx \frac{n}{B} \times C_{\text{store_root}} \quad (4)$$

In these equations, C_{merkle} represents the total gas cost for processing n reports using aggregation. The variable n denotes the number of reports and B represents the batch size. The component $C_{\text{store_root}}$ represents the average cost of storing a single Merkle Root into the smart contract, including the overhead of executing the `commitBatch` function and the permanent storage operations within the `BatchCommit` structure. Theoretically, when $B > 1$, gas consumption decreases significantly because only a single fixed-size hash is recorded per batch. However, this efficiency introduces additional off-chain computation and requires a Merkle Proof-based verification mechanism when individual verification is needed. This trade-off between gas reduction and verification complexity is analyzed in the experimental stage.

II.5.3 Algorithm 3: Storage Optimization (Packed Struct + Enum Encoding)

This approach reduces gas consumption not by decreasing the number of transactions, but by compacting the representation of data stored in each transaction [48]. Dynamic data types such as string are replaced with more efficient fixed-length and numeric types: categorical attributes are encoded as `uint8`, while identifiers and hashes are represented as `bytes32`. Nine `uint8` fields are then packed into a single `uint128` variable using bitwise operations, allowing multiple attributes to share a single 32-byte storage slot. The optimized structure is shown in Algorithm 8.

```

struct PackedAuditLog { bytes32 reportCode; bytes32 dataHash; uint256 timestamp; address actor; uint128 packedEnums; }

```

Algorithm 8: PackedAuditLog Structure in StorageOptimized Smart Contract.

Source: Authors, (2026).

The number of transactions remains linear with respect to the number of reports $O(n)$, but each entry uses fewer storage slots compared with a dynamic string-based representation. The four primary components (`reportCode`, `dataHash`, `timestamp`, and `actor`) occupy individual slots, while nine categorical attributes share a single slot through `uint128`, thereby reducing the per-transaction cost constant. The total gas cost is formulated in Equation (5):

$$C_{\text{packed}} = \sum_{i=1}^n \left(G_{\text{base}} + G_{\text{calldata},i}^{\text{fixed}} + G_{\text{storage},i}^{\text{optimized}} \right) \quad (5)$$

Which can be approximately simplified into Equation (6):

$$C_{\text{packed}} \approx n \times C_{\text{per_tx}}^{\text{optimized}} \quad (6)$$

In this formulation, C_{packed} represents the total gas cost for n reports with the optimized structure. The component G_{base} still represents the base transaction cost. The value $G_{\text{calldata},i}^{\text{fixed}}$ indicates a lower calldata cost because all parameters use fixed-length types. Meanwhile, $G_{\text{storage},i}^{\text{optimized}}$ represents the reduced SSTORE operation cost due to the elimination of dynamic storage allocation and the implementation of a storage packing mechanism. Furthermore, $C_{\text{per_tx}}^{\text{optimized}}$ is the estimated average cost per transaction after the data structure optimization is applied.

II.5.4 Algorithm 4: Conditional Commit (Threshold-Based Batching)

This approach implements an off-chain buffering mechanism, in which reports are not immediately submitted to the blockchain but are first accumulated until a specified threshold value is reached [49], [50]. Once the number of reports in the buffer meets this threshold, the system submits a single commit transaction that records an aggregated hash representing all reports within the batch. Consequently, the number of on-chain transactions is reduced proportionally to the threshold value applied.

The metadata for each batch is recorded in the BatchRecord structure as illustrated in Algorithm 9, which contains the batch identifier, the aggregated hash, the number of reports included, the range of report identifiers, the threshold value applied at the time of commit, the recording timestamp, and the address of the transaction sender.

```
struct BatchRecord {
    uint256 batchId; bytes32 batchHash; uint256 reportCount; uint256 fromId; uint256 toId; uint256 threshold; uint256 timestamp; address committer; }
```

Algorithm 9: BatchRecord Structure in ConditionalCommit Smart Contract.

Source: Authors, (2026).

The initial threshold value is configured through the contract constructor as shown in Algorithm 10, which assigns the contract owner, initializes the total number of committed reports, and stores the initial threshold value as a system control parameter.

```
constructor(uint256 _initialThreshold) { owner = msg.sender; totalReportsCommitted = 0; currentThreshold = _initialThreshold; }
```

Algorithm 10: Constructor for Initial Threshold Configuration.

Source: Authors, (2026).

The procedural flow of the commit mechanism is presented in Algorithm 12.

Input: <i>n</i> reports and threshold value <i>T</i>
Output: List of commit transaction proofs
1. Prepare an empty buffer to store report hashes 2. For each report: a. Compute the report hash as an integrity marker b. Store the hash in the buffer c. If the number of elements in the buffer reaches <i>T</i>: - Concatenate all hashes in the buffer - Compute a single aggregated hash from the concatenated data - Submit the aggregated hash together with batch metadata to the smart contract - Wait for transaction confirmation and store the transaction proof and then clear the buffer 3. After all reports have been processed, if any elements remain in the buffer, perform a final commit 4. Return all transaction proofs

Algorithm 12: ConditionalCommit Procedure.

Source: Authors, (2026).

From a computational perspective, each report is processed only once to generate its hash, resulting in an off-chain complexity of $O(n)$. However, the number of on-chain transactions decreases to $O(n/T)$, where T denotes the threshold value. Experimental evaluation for $n = 1000$ reports under several threshold values is presented in Table 5, demonstrating that increasing T reduces transaction frequency but increases commit latency, since reports must wait until the buffer reaches the defined threshold.

Table 5: Evaluated Threshold Parameters.

Threshold	On-chain Transactions (n=1000)	Reduction Ratio	Characteristics
5	200	5:1	More frequent commits with relatively low latency
10	100	10:1	Balanced trade-off between gas savings and responsiveness
25	40	25:1	High gas savings
50	20	50:1	Maximum gas savings with higher commit latency

Source: Authors, (2026).

The total gas cost is formulated in Equation (7):

$$C_{\text{conditional}} = \left\lceil \frac{n}{T} \right\rceil \times C_{\text{commit}} \quad (7)$$

If n is significantly larger than T , the total cost can be approximated as shown in Equation (8):

$$C_{\text{conditional}} \approx \frac{n}{T} \times C_{\text{commit}} \quad (8)$$

In these expressions, $C_{\text{conditional}}$ represents the total gas cost required to process n reports using the threshold-based mechanism. The variable n denotes the number of reports, T represents the threshold value per batch, and C_{commit} corresponds to the average gas cost of a single commit transaction that records an aggregated hash on the blockchain. The optimal threshold value is determined by calculating the effective gas per report, defined as the ratio between the total gas consumed and the number of processed reports. The value of T that produces the lowest average cost is selected as the optimal configuration, while also considering the trade-off between gas efficiency and increased commit latency.

III. RESULTS AND DISCUSSIONS

This chapter presents the performance evaluation results of various gas fee optimization approaches in the implementation of smart contracts for a blockchain-based sexual violence reporting system. The dataset used in the experiment consists of 1,000 simulated reports with an attribute structure as shown in Table 6, representing various report characteristics such as victim type, level of concern, incident location, and system action metadata. The experiments were conducted using a CPU-only computational environment without GPU acceleration, with hardware specifications consisting of an AMD Ryzen 5 6600H (12 CPUs) processor operating at approximately 3.3 GHz and 16 GB of RAM.

In this evaluation scenario, the Conditional Commit approach was implemented using a batching mechanism with a threshold of 50 reports, where a commit to the blockchain is executed once the number of reports in the buffer reaches the defined threshold. Subsequently, the performance of the Baseline, Merkle Tree Aggregation, Storage Optimized, and Conditional Commit methods was analyzed based on several key metrics, namely total gas consumption, transaction cost (ETH), transaction latency, and system execution time, across various report quantity scenarios.

Table 6: Sample Structure of Sexual Violence Reporting Dataset Used in Experiments.

No	Report Code	Status Darurat	Korban	Tingkat Kekhawatiran	Gender Korban	Pelaku Kekerasan	Tanggal Kejadian	Lokasi Kejadian	Lokasi Detail	Ringkasan Kejadian (detail singkat)	Usia Korban	Disabilitas	Jenis Disabilitas	Action Type	Actor Role	Email Korban
1	PPKPT-100000	tidak	saya	sangat	lakilaki	orang_tidak_dikenal	2025-10-11	Masjid	Kantin depan	Perundungan daring melalui grup medsos kampus	46-55	ya	kesulitan melihat	DISPUTE	SYSTEM	korban0@gmail.com
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1000	PPKPT-100999	tidak	saya	khawatir	lakilaki	mantan_pacar	2025-12-30	Luar Kampus	Lantai 2 dekat perpustakaan	Pelecehan verbal di area kampus saat jam kuliah	18-25	ya	kesulitan berkomunikasi	CREATE	USER	pelapor999@student.ac.id

Source: Authors, (2026).

The experimental results demonstrate clear performance differences between the baseline approach and the three proposed optimization methods in terms of gas consumption, transaction cost, latency, and execution time. As shown in Table 7, the baseline method consistently produces the highest gas consumption across all report quantity scenarios. The total gas increases from 38,430,050 gas for 50 reports to 770,899,033 gas for 1,000 reports, exhibiting a nearly linear growth pattern as the number of reports increases. This condition occurs because each report is processed as an independent on-chain transaction and stores several data fields in the smart contract storage. In the Ethereum Virtual Machine (EVM), storage write operations through the SSTORE instruction incur relatively high gas costs, particularly when the stored data has large size or uses dynamic data types. Consequently, as the number of processed reports increases, the number of storage operations also increases proportionally, directly affecting the total gas consumption and transaction cost. Statistical analysis using the Kruskal–Wallis test indicates that the differences between methods in the gas per transaction metric are significant across all scenarios; for example, in the 100-report scenario, a value of $H = 166.93$ with $p < 0.01$ was obtained. The Merkle Tree Aggregation method demonstrates a substantial reduction in gas consumption compared with the baseline. In this approach, multiple reports are first combined off-chain to form a Merkle Tree structure, after which only the Merkle Root value is recorded on the blockchain through a single commit transaction.

Through this mechanism, the number of on-chain transactions is reduced because a single transaction can represent multiple reports simultaneously. Data in Table 7 show that the total gas required for 1,000 reports reaches only 20,492,732 gas, which is significantly lower than the baseline. The Mann–Whitney test also indicates that the difference in gas consumption between the baseline and the Merkle Tree method is statistically significant across various scenarios; for instance, in the 500-report scenario, $p < 0.01$. The reduction in the number of transactions also contributes to lower latency and execution time, since fewer transactions need to be processed and confirmed by the blockchain network. The Storage Optimized approach reduces gas consumption through modifications to the data structure stored within the smart contract. This method replaces dynamic data types such as string with fixed-length types such as bytes32 and utilizes storage packing techniques to reduce the number of storage slots used. This optimization decreases the cost of data writing because fewer storage slots need to be updated. Experimental results indicate that the total gas consumption in the 1,000-report scenario is 177,566,016 gas, which is lower than the baseline but still higher than aggregation-based methods. Statistical testing also shows that the difference between the baseline and Storage Optimized approaches is significant in nearly all scenarios; for example, in the 100-report scenario, $p < 0.01$. This outcome arises from the characteristic of the method that continues to process each report as an individual transaction, resulting in the number of transactions still increasing linearly with the number of reports.

Table 7: Total Gas Consumption, Transaction Cost (ETH), Latency, and Execution Time for Each Scenario (50, 100, 500, and 1000) and Optimization Method.

Scenario	Method	Total Gas	Total Cost (ETH)	Total Latency (ms)	Total Exec Time (s)
50	Baseline	38,430,050	0.77	7066.71	7.07
	Merkle Tree	1,042,788	0.02	271.20	0.27
	Storage Optimized	9,303,068	0.19	2004.89	2.00
	Conditional Commit	196,487	0.00	44.49	0.04
100	Baseline	76,792,122	1.54	14685.12	14.69
	Merkle Tree	2,048,288	0.04	453.85	0.45
	Storage Optimized	17,756,628	0.36	4121.87	4.12
	Conditional Commit	412,886	0.01	118.53	0.12
500	Baseline	384,231,209	7.68	71282.51	71.28
	Merkle Tree	10,492,592	0.21	2315.19	2.32
	Storage Optimized	91,348,056	1.83	21870.83	21.87

Scenario	Method	Total Gas	Total Cost (ETH)	Total Latency (ms)	Total Exec Time (s)
1000	Conditional Commit	2,144,174	0.04	524.33	0.52
	Baseline	770,899,033	15.42	143722.60	143.72
	Merkle Tree	20,492,732	0.41	4640.03	4.64
	Storage Optimized	177,566,016	3.55	45480.59	45.48
	Conditional Commit	4,308,404	0.09	1077.91	1.08

Source: Authors, (2026).

Table 8: Per-Transaction Average Gas Usage, Cost (ETH), Latency, and Execution Time (Mean $\pm$ Standard Deviation) Across Methods and Scenarios.

Scenario	Method	Average Gas/Tx	Average Cost ETH	Average Latency (ms)	Average Exec Time/Tx (s)
50	Baseline	768,601 $\pm$ 15,724	0.02 $\pm$ 0.00	141.33 $\pm$ 17.78	0.14 $\pm$ 0.02
	Merkle Tree	208,558 $\pm$ 11,409	0.00 $\pm$ 0.00	54.24 $\pm$ 22.42	0.05 $\pm$ 0.02
	Storage Optimized	186,061 $\pm$ 397	0.00 $\pm$ 0.00	40.10 $\pm$ 3.99	0.04 $\pm$ 0.00
	Conditional Commit	196,487 $\pm$ 0	0.00 $\pm$ 0.00	44.49 $\pm$ 0.00	0.04 $\pm$ 0.00
100	Baseline	767,921 $\pm$ 13,418	0.02 $\pm$ 0.00	146.85 $\pm$ 19.23	0.15 $\pm$ 0.02
	Merkle Tree	204,829 $\pm$ 9,347	0.00 $\pm$ 0.00	45.38 $\pm$ 6.87	0.05 $\pm$ 0.01
	Storage Optimized	177,566 $\pm$ 8,592	0.00 $\pm$ 0.00	41.22 $\pm$ 6.13	0.04 $\pm$ 0.01
	Conditional Commit	206,443 $\pm$ 14,080	0.00 $\pm$ 0.00	59.27 $\pm$ 35.62	0.06 $\pm$ 0.04
500	Baseline	768,462 $\pm$ 13,373	0.02 $\pm$ 0.00	142.57 $\pm$ 14.89	0.14 $\pm$ 0.01
	Merkle Tree	209,852 $\pm$ 8,177	0.00 $\pm$ 0.00	46.30 $\pm$ 7.38	0.05 $\pm$ 0.01
	Storage Optimized	182,696 $\pm$ 6,847	0.00 $\pm$ 0.00	43.74 $\pm$ 7.28	0.04 $\pm$ 0.01
	Conditional Commit	214,417 $\pm$ 6,300	0.00 $\pm$ 0.00	52.43 $\pm$ 5.38	0.05 $\pm$ 0.01
1000	Baseline	770,899 $\pm$ 14,891	0.02 $\pm$ 0.00	143.72 $\pm$ 12.31	0.14 $\pm$ 0.01
	Merkle Tree	204,927 $\pm$ 9,020	0.00 $\pm$ 0.00	46.40 $\pm$ 9.39	0.05 $\pm$ 0.01
	Storage Optimized	177,566 $\pm$ 8,554	0.00 $\pm$ 0.00	45.48 $\pm$ 6.14	0.05 $\pm$ 0.01
	Conditional Commit	215,420 $\pm$ 4,456	0.00 $\pm$ 0.00	53.90 $\pm$ 4.51	0.05 $\pm$ 0.00

Source: Authors, (2026).

The Conditional Commit method produces the lowest total gas consumption among all evaluated approaches. This mechanism collects several reports within a buffer until a specified threshold is reached before performing a commit to the blockchain. Each commit stores an aggregated hash representing a batch of reports, enabling a single transaction to record multiple reports simultaneously. By reducing the frequency of on-chain transactions, this method significantly decreases overall gas consumption and shortens the system execution time. In the 1,000-report scenario, the total gas consumption is 4,308,404 gas with an execution time of approximately 1.08 seconds, which is substantially lower than the baseline that requires 143.72 seconds. Statistical testing results also indicate that latency differences among methods are significant; for instance, in the 100-report scenario, a value of $H = 159.70$ with $p < 0.01$ was obtained, demonstrating that reducing the number of transactions has a substantial impact on improving the efficiency of blockchain system performance. The average per-transaction metric analysis in Table 8 provides additional insights into the execution characteristics of each method. The baseline approach exhibits an average gas consumption of approximately 768,000 to 770,000 gas per transaction, with relatively small variation across scenarios, indicating that the transaction operation structure remains relatively consistent. The Merkle Tree method reduces this average value to approximately 204,000 to 209,000 gas, because the transaction only stores batch metadata and the Merkle Root. The Storage Optimized approach produces an average of approximately 177,000 to 186,000 gas per transaction as a result of improved storage structure efficiency. The Conditional Commit method has an average of approximately 196,000 to 215,000 gas per transaction, slightly higher than Storage Optimized because each commit transaction stores additional metadata; however, the significantly smaller number of transactions ensures that the overall system gas consumption remains low.

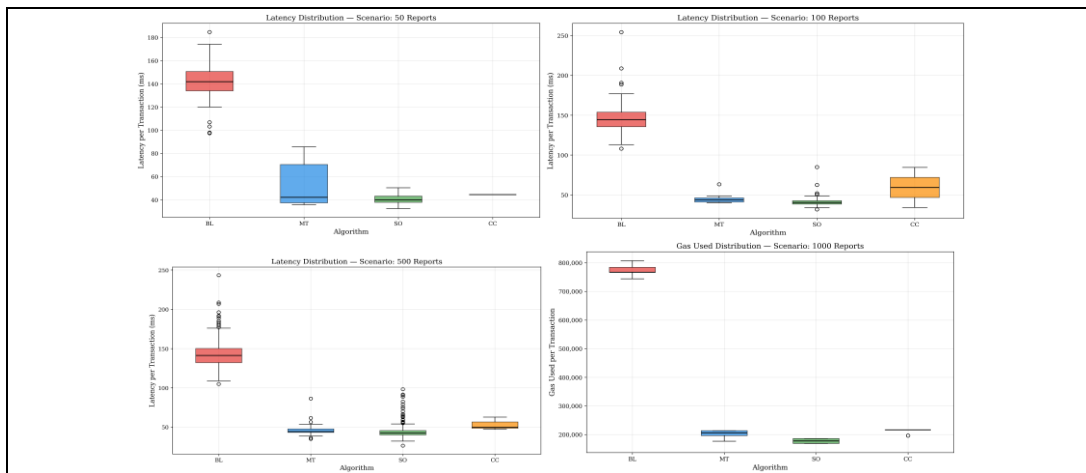


Figure 4: Boxplot of Transaction Latency Distribution (ms) Across Optimization Methods.

Source: Authors, (2026).

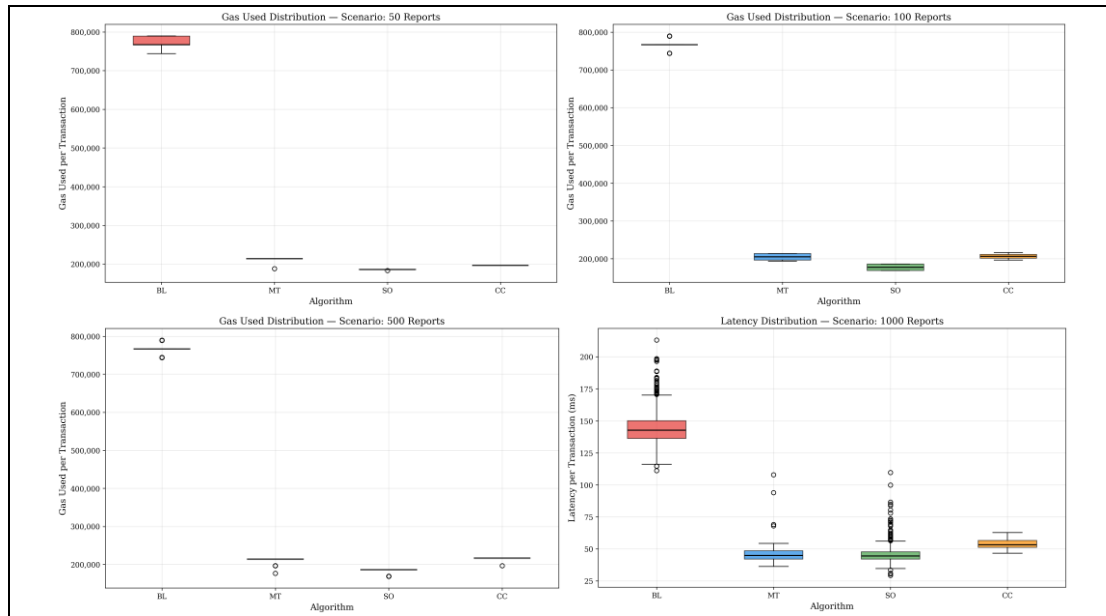


Figure 5: Boxplot of Gas Consumption Distribution per Transaction Across Optimization Methods.
Source: Authors, (2026).

The distribution of transaction latency visualized in Figure 4 shows distinct performance patterns among the evaluated methods. The baseline approach exhibits a median latency of approximately 140–145 ms with a wider interquartile range compared to the other methods, indicating relatively higher variability in execution time. This variation is associated with the larger calldata size and the greater number of storage operations performed in each transaction. The Merkle Tree and Storage Optimized methods demonstrate lower median latency values in the range of approximately 40–46 ms, with more concentrated distributions, reflecting lighter transaction execution processes. The Conditional Commit approach has a slightly higher median latency than these two methods, around 52–54 ms, yet it remains lower than the baseline and shows relatively small variation as the number of batches increases. The distribution of gas consumption per transaction presented in Figure 5 indicates that the baseline method has the highest median value, approximately 766 thousand to 767 thousand gas. The Merkle Tree method reduces this value to approximately 205 thousand to 213 thousand gas per transaction, whereas the Storage Optimized approach shows a median of approximately 177 thousand to 186 thousand gas with relatively small variation. The Conditional Commit method has a median of approximately 196 thousand to 216 thousand gas, reflecting the storage of batch metadata within each commit transaction. The baseline approach shows a value of approximately 768 thousand gas per report across all scenarios, while Storage Optimized reduces it to approximately 177 thousand to 186 thousand gas. The Merkle Tree method produces a consumption of around 20 thousand gas per report because the transaction cost is distributed across all reports within a batch. The Conditional Commit approach demonstrates the lowest value, approximately 3,930 to 4,308 gas per report, since a single commit transaction represents multiple reports simultaneously.

This study provides an empirical contribution to the understanding of factors influencing the efficiency of blockchain-based reporting systems. By comparing several optimization approaches across different report quantity scenarios, the study demonstrates that strategies aimed at reducing the number of on-chain transactions can produce a greater impact on gas efficiency than data structure optimization alone. The evaluation conducted through metrics such as total gas consumption, average per-transaction metrics, and performance distribution analysis offers a more comprehensive overview of the performance characteristics of each method. This study has several limitations, primarily related to gas costs on the Ethereum network. Although the proposed optimization techniques significantly reduce gas consumption compared to the baseline approach, blockchain transaction costs remain relatively high, particularly as the number of reports increases. Furthermore, the experiments were conducted in a controlled testing environment, which does not fully represent the dynamic conditions of public blockchain networks, such as gas price fluctuations and transaction congestion levels. The evaluation in this study also focuses on primary metrics such as gas consumption, transaction cost, latency, and execution time, while other aspects that may influence cost efficiency, such as off-chain computation overhead and batch management, have not yet been analyzed in depth. Future research may explore more adaptive optimization strategies to mitigate the impact of high gas costs. One potential approach is the implementation of adaptive batching, which dynamically adjusts batch size or commit timing based on network conditions and gas prices. In addition, integration with layer-2 scaling solutions, such as rollups or sidechains, may further reduce transaction costs significantly [51]. Further evaluation on public blockchain networks with varying gas price conditions is necessary to obtain a more realistic understanding of system performance in real-world implementations.

IV. CONCLUSIONS

This study evaluated several smart contract optimization strategies to reduce gas consumption in a blockchain-based mobile reporting system for sexual violence cases on the Ethereum network. The findings indicate that the efficiency of blockchain applications is strongly influenced by how data storage and transaction submission are designed. Approaches that store detailed information directly on-chain result in high computational costs due to intensive storage operations, whereas strategies that minimize storage usage or reduce the frequency of on-chain transactions significantly improve scalability. In particular, aggregation and threshold-based batching mechanisms enable multiple reports to be represented through compact cryptographic commitments while preserving data integrity.

Scientifically, this research contributes a comparative experimental framework that integrates storage-level optimization, cryptographic aggregation, and transaction batching within a single evaluation environment, providing empirical evidence on how different design strategies affect gas efficiency and execution performance. Practically, the results provide guidance for developers and system architects in designing blockchain-based audit logging systems that remain economically feasible on public blockchain infrastructures. Nevertheless, the study is limited to experiments conducted in a controlled environment and focuses primarily on gas efficiency metrics without evaluating broader aspects such as security trade-offs or real network variability. Future studies should examine adaptive batching mechanisms, cross-platform blockchain implementations, and security implications of off-chain aggregation to further improve the scalability and reliability of decentralized reporting systems.

VI. AUTHOR'S CONTRIBUTION

Conceptualization: Michael Angello Qadosy Riyadi, Sulthonika Mahfudz Al Mujahidin, Nur Alifia Rustan, A'isyah Belqis Febi Aulia, Mastuty Ayu Ningtyas and Khodijah Amiroh.

Methodology: Michael Angello Qadosy Riyadi, Sulthonika Mahfudz Al Mujahidin, Nur Alifia Rustan, A'isyah Belqis Febi Aulia, Mastuty Ayu Ningtyas and Khodijah Amiroh.

Investigation: Michael Angello Qadosy Riyadi, Sulthonika Mahfudz Al Mujahidin, Nur Alifia Rustan, A'isyah Belqis Febi Aulia, Mastuty Ayu Ningtyas and Khodijah Amiroh.

Discussion of results: Michael Angello Qadosy Riyadi, Sulthonika Mahfudz Al Mujahidin, Nur Alifia Rustan, A'isyah Belqis Febi Aulia, Mastuty Ayu Ningtyas and Khodijah Amiroh.

Writing – Original Draft: Michael Angello Qadosy Riyadi, Sulthonika Mahfudz Al Mujahidin, Nur Alifia Rustan, A'isyah Belqis Febi Aulia, Mastuty Ayu Ningtyas and Khodijah Amiroh.

Writing – Review and Editing: Michael Angello Qadosy Riyadi, Sulthonika Mahfudz Al Mujahidin, Nur Alifia Rustan, A'isyah Belqis Febi Aulia, Mastuty Ayu Ningtyas and Khodijah Amiroh.

Resources: Michael Angello Qadosy Riyadi, Sulthonika Mahfudz Al Mujahidin, Nur Alifia Rustan, A'isyah Belqis Febi Aulia, Mastuty Ayu Ningtyas and Khodijah Amiroh.

Supervision: Michael Angello Qadosy Riyadi, Sulthonika Mahfudz Al Mujahidin, Nur Alifia Rustan, A'isyah Belqis Febi Aulia, Mastuty Ayu Ningtyas and Khodijah Amiroh.

Approval of the final text: Michael Angello Qadosy Riyadi, Sulthonika Mahfudz Al Mujahidin, Nur Alifia Rustan, A'isyah Belqis Febi Aulia, Mastuty Ayu Ningtyas and Khodijah Amiroh.

VII. REFERENCES

- [1] S. Dong, K. Abbas, M. Li, and J. Kamruzzaman, "Blockchain technology and application: an overview," *PeerJ Computer Science*, vol. 9, p. e1705, 2023, doi: 10.7717/peerj-cs.1705.
- [2] M. Krichen, M. Ammi, A. Mihoub, and M. Almutiq, "Blockchain for Modern Applications: A Survey," *Sensors*, vol. 22, no. 14, p. 5274, 2022, doi: 10.3390/s22145274.
- [3] G. Habib, S. Sharma, S. Ibrahim, I. Ahmad, S. Qureshi, and M. Ishfaq, "Blockchain Technology: Benefits, Challenges, Applications, and Integration of Blockchain Technology with Cloud Computing," *Future Internet*, vol. 14, no. 11, p. 341, 2022, doi: 10.3390/fi14110341.
- [4] N. Machado and C. M. Taboada Rodriguez, "A Logistics 5.0 maturity model: a human-centric and sustainable approach for the supply chain of the future," *ITEGAM-JETIA*, vol. 11, no. 51, pp. 164–170, 2025, doi: 10.5935/jetia.v11i51.1407.
- [5] T. Kazeem et al., "Risk Management and Governance in Blockchain-Based Digital Identity Projects: A Business Analysis and Project Management Framework," *ITEGAM-JETIA*, vol. 11, no. 54, pp. 172–180, 2025, doi: 10.5935/jetia.v11i54.1710.
- [6] D. L. Fekete and A. Kiss, "Toward Building Smart Contract-Based Higher Education Systems Using Zero-Knowledge Ethereum Virtual Machine," *Electronics*, vol. 12, no. 3, p. 664, 2023, doi: 10.3390/electronics12030664.
- [7] F. Cassez, J. Fuller, M. K. Ghale, D. J. Pearce, and H. M. A. Quiles, "Formal and Executable Semantics of the Ethereum Virtual Machine in Dafny," in *Formal Methods. FM 2023*, M. Chechik, J. P. Katoen, and M. Leucker, Eds., in *Lecture Notes in Computer Science*, vol. 14000. Cham: Springer, 2023, pp. 571–583. doi: 10.1007/978-3-031-27481-7_32.
- [8] C. E. Quintero-Narvaez and R. Monroy-Borja, "Zero-Knowledge Proofs for Questionnaire Result Verification in Smart Contracts," in *2023 Mexican International Conference on Computer Science (ENC)*, Guanajuato, Guanajuato, Mexico: IEEE, 2023, pp. 1–5. doi: 10.1109/ENC60556.2023.10508637.
- [9] B. K. Meister and H. C. W. Price, "Gas fees on the Ethereum blockchain: from foundations to derivative valuations," *Frontiers in Blockchain*, vol. 7, p. 1462666, 2024, doi: 10.3389/fbloc.2024.1462666.
- [10] P. Sharma, R. Jindal, and M. D. Borah, "A review of smart contract-based platforms, applications, and challenges," *Cluster Computing*, vol. 26, pp. 395–421, 2023, doi: 10.1007/s10586-021-03491-1.
- [11] M. M. A. Khan, H. M. A. Sarwar, and M. Awais, "Gas consumption analysis of Ethereum blockchain transactions," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 4, p. e6679, 2022, doi: 10.1002/cpe.6679.
- [12] P. Ahirao, Y. Borse, D. Patole, S. Agarwal, M. Abhyankar, and K. Nair, "Comprehensive Analysis of Mitigation Strategies for Denial-of-Service Attacks on Smart Contracts," in *2025 IEEE International Conference on Blockchain and Distributed Systems Security (ICBDS)*, Kolhapur, India: IEEE, 2025, pp. 1–6. doi: 10.1109/ICBDS67396.2025.11376964.
- [13] H. A. Janjua, L. Yue, and S. Hayat, "Smart Scan: An Approach to Detect Denial of Service Vulnerability in Ethereum Smart Contracts," in *Proceedings of the 2023 7th International Conference on Electronic Information Technology and Computer Engineering (EITCE '23)*, New York, NY, USA: Association for Computing Machinery, 2023, pp. 1572–1578. doi: 10.1145/3650400.3650663.

- [14] M. T. Ta and T. Q. Do, "A study on gas cost of ethereum smart contracts and performance of blockchain on simulation tool," *Peer-to-Peer Networking and Applications*, vol. 17, pp. 200–212, 2024, doi: 10.1007/s12083-023-01598-3.
- [15] S. Porkodi and D. Kesavaraja, "Escalating Gas Cost Optimization in Smart Contract," *Wireless Personal Communications*, vol. 136, pp. 35–59, 2024, doi: 10.1007/s11277-024-11066-7.
- [16] M. Pacheco, G. Oliva, G. K. Rajbahadur, and A. Hassan, "Is My Transaction Done Yet? An Empirical Study of Transaction Processing Times in the Ethereum Blockchain Platform," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, p. Article 59, 2023, doi: 10.1145/3549542.
- [17] A. Vacca, M. Fredella, A. Di Sorbo, C. A. Visaggio, and G. Canfora, "An empirical investigation on the trade-off between smart contract readability and gas consumption," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC '22)*, New York, NY, USA: Association for Computing Machinery, 2022, pp. 214–224. doi: 10.1145/3524610.3529157.
- [18] A. S. Paramitha and M. Tarigan, "Analysis of Gas Fee Patterns in Blockchain Transactions - A Case Study on Ethereum Smart Contracts," *Journal of Current Research in Blockchain*, vol. 2, no. 3, pp. 180–189, 2025, doi: 10.47738/jcrb.v2i3.41.
- [19] E. Ni, E. Knight, and M. Gerstein, "Scalable and efficient on-chain data management in blockchain for large biomedical data," *Journal of Biomedical Informatics*, vol. 165, p. 104818, 2025, doi: <https://doi.org/10.1016/j.jbi.2025.104818>.
- [20] A. B. M. Salem and M. J. Aqel, "Analyzing Transaction Fee Patterns and Their Impact on Ethereum Blockchain Efficiency," *Journal of Current Research in Blockchain*, vol. 2, no. 4, pp. 228–243, 2025, doi: 10.47738/jcrb.v2i4.46.
- [21] D. Koutmos, "Network Activity and Ethereum Gas Prices," *Journal of Risk and Financial Management*, vol. 16, no. 10, p. 431, 2023, doi: 10.3390/jrfm16100431.
- [22] X. Wang and F. Xu, "The Value of Smart Contract in Trade Finance," *Manufacturing & Service Operations Management*, vol. 25, no. 6, pp. 2056–2073, 2023, doi: 10.1287/msom.2022.1126.
- [23] N. Afraz, F. Wilhelm, H. Ahmadi, and M. Ruffini, "Blockchain and Smart Contracts for Telecommunications: Requirements vs. Cost Analysis," *IEEE Access*, vol. 11, pp. 95653–95666, 2023, doi: 10.1109/ACCESS.2023.3309423.
- [24] J. Huang, K. Zhou, A. Xiong, and D. Li, "Smart Contract Vulnerability Detection Model Based on Multi-Task Learning," *Sensors*, vol. 22, no. 5, p. 1829, 2022, doi: 10.3390/s22051829.
- [25] Z. Zhao, J. Li, Z. Su, and Y. Wang, "GaSaver: A Static Analysis Tool for Saving Gas," *IEEE Transactions on Sustainable Computing*, vol. 8, no. 2, pp. 257–267, 2023, doi: 10.1109/TSUSC.2022.3221444.
- [26] A. Kuhlman and A. Wicaksana, "Smart contract optimization for gas fee reduction with static solidity optimizer," *Discover Applied Sciences*, vol. 7, p. 888, 2025, doi: 10.1007/s42452-025-07537-6.
- [27] A. R. Raza, M. Hanzala, K. Mahmood, S. Tahir, S. Khalid, and A. Masood, "Optimizing Gas Consumption in Ethereum Smart Contracts: A Static Analysis Approach," in *2025 International Conference on Communication Technologies (ComTech)*, Rawalpindi, Pakistan: IEEE, 2025, pp. 1–6. doi: 10.1109/ComTech65062.2025.11034442.
- [28] S. Khanzadeh and M. H. Alalfi, "SoIOSphere: A Framework for Gas Optimization in Solidity Smart Contracts," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, Rovaniemi, Finland: IEEE, 2024, pp. 35–45. doi: 10.1109/SANER-C62648.2024.00010.
- [29] S. Gupta, S. Singh, Snehlata, and V. K. Dwivedi, "Gas-Efficient Patterns for Smart Contracts in Solidity: Optimization Techniques and Comparative Analysis," in *2024 4th International Conference on Innovative Sustainable Computational Technologies (CISCT)*, Dehradun, India: IEEE, 2024, pp. 1–6. doi: 10.1109/CISCT62494.2024.11134254.
- [30] C. C. Escobar, S. Roy, O. P. Kreidl, A. Dutta, and L. Bölöni, "Toward a Green Blockchain: Engineering Merkle Tree and Proof of Work for Energy Optimization," *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 3847–3857, 2022, doi: 10.1109/TNSM.2022.3219494.
- [31] H. V. A. Le, Q. D. N. Nguyen, N. Tadashi, and T. H. Tran, "Blockchain-Based Decentralized Identity Management System with AI and Merkle Trees," *Computers*, vol. 14, no. 7, p. 289, 2025, doi: 10.3390/computers14070289.
- [32] M. Lakshmanan and G. S. Anandha Mala, "Merkle tree-blockchain-assisted privacy preservation of electronic medical records on offering medical data protection through hybrid heuristic algorithm," *Knowledge and Information Systems*, vol. 66, pp. 481–509, 2024, doi: 10.1007/s10115-023-01937-z.
- [33] Z. Liu, L. Ren, Y. Feng, S. Wang, and J. Wei, "Data Integrity Audit Scheme Based on Quad Merkle Tree and Blockchain," *IEEE Access*, vol. 11, pp. 59263–59273, 2023, doi: 10.1109/ACCESS.2023.3240066.
- [34] S. Park et al., "Impact of EIP-4844 on Ethereum: Consensus Security, Ethereum Usage, Rollup Transaction Dynamics, and Blob Gas Fee Markets," *IEEE Transactions on Network Science and Engineering*, vol. 13, pp. 1294–1310, 2026, doi: 10.1109/TNSE.2025.3594281.
- [35] Y. Fan, B. Gao, D. Sheng, Z. Hu, and X. Ding, "Batch Transaction Processing for Adaptive Sharding Blockchain-Enabled Edge Computing," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 20, no. 4, p. Article 31, 2025, doi: 10.1145/3764927.
- [36] N. Maryani, M. Muchlish, R. Mulyadi, and N. Solehah, "Blockchain-Based Audit Trails: Improving Transparency and Fraud Detection in Digital Accounting Systems," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 17, no. 1, 2026, doi: 10.14569/IJACSA.2026.0170162.
- [37] F. Casino, T. K. Dasaklis, and C. Patsakis, "A systematic literature review of blockchain-based applications: Current status, classification and open issues," *Telematics and Informatics*, vol. 36, pp. 55–81, 2019, doi: 10.1016/j.tele.2018.11.006.
- [38] A. Alketbi, Q. Nasir, and M. A. Talib, "Blockchain for government services — Use cases, security benefits and challenges," in *2018 15th Learning and Technology Conference (L&T)*, Jeddah, Saudi Arabia: IEEE, 2018, pp. 112–119. doi: 10.1109/LT.2018.8368494.
- [39] A. Akhtar, B. Shafiq, J. Vaidya, A. Afzal, S. Shamail, and O. Rana, "Blockchain Based Auditable Access Control for Distributed Business Processes," *Proceedings International Conference on Distributed Computing Systems*, pp. 12–22, 2020, doi: 10.1109/ICDCS47774.2020.00015.

- [40] H. M. Zangana, A. khalid Mohammed, and S. R. M. Zeebaree, "Systematic Review of Decentralized and Collaborative Computing Models in Cloud Architectures for Distributed Edge Computing," *Sistemasi: Jurnal Sistem Informasi*, vol. 13, no. 4, 2024, doi: 10.32520/stmsi.v13i4.4169.
- [41] L. Shekhtman and E. Waisbard, "EngraveChain: A Blockchain-Based Tamper-Proof Distributed Log System," *Future Internet*, vol. 13, no. 6, p. 143, 2021, doi: 10.3390/fi13060143.
- [42] A. Ahmad, M. Saad, M. Bassiouni, and A. Mohaisen, "Towards Blockchain-Driven, Secure and Transparent Audit Logs," in *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous '18)*, New York, NY, USA: Association for Computing Machinery, 2018, pp. 443–448. doi: 10.1145/3286978.3286985.
- [43] S. Azimi, A. Golzari, N. Ivaki, and N. Laranjeiro, "A systematic review on smart contracts security design patterns," *Empirical Software Engineering*, vol. 30, p. 95, 2025, doi: 10.1007/s10664-025-10646-w.
- [44] C. Dannen, "Solidity Programming," in *Introducing Ethereum and Solidity*, Berkeley, CA: Apress, 2017, pp. 69–88. doi: 10.1007/978-1-4842-2535-6_4.
- [45] P. Xu, Y. Sui, and M. Staples, "Towards Source Mapping for Zero-Knowledge Smart Contracts: Design and Preliminary Evaluation," in *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA Companion '25)*, New York, NY, USA: Association for Computing Machinery, 2025, pp. 200–209. doi: 10.1145/3713081.3732932.
- [46] O. Kuznetsov, A. Rusnak, A. Yezhov, D. Kanonik, K. Kuznetsova, and O. Domin, "Efficient and Universal Merkle Tree Inclusion Proofs via OR Aggregation," *Cryptography*, vol. 8, no. 3, p. 28, 2024, doi: 10.3390/cryptography8030028.
- [47] J. Davies, "Enhanced scalability and privacy for blockchain data using Merklized transactions," *Frontiers in Blockchain*, vol. 6, p. 1222614, 2024, doi: 10.3389/fbloc.2023.1222614.
- [48] S. Khanzadeh, N. Samreen, and M. H. Alalfi, "Optimizing Gas Consumption in Ethereum Smart Contracts: Best Practices and Techniques," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, Chiang Mai, Thailand: IEEE, 2023, pp. 300–309. doi: 10.1109/QRS-C60940.2023.00056.
- [49] K. S. Weedagamaarachchi, H. D. Vithanage, H. M. S. N. Dehipola, K. D. R. Manditha, V. Jayasinghearachchi, and J. Perera, "Optimizing Gas Efficiency in Ethereum Smart Contracts Using Design Pattern Refinement and Complexity Reduction," in *2025 7th International Conference on Advancements in Computing (ICAC)*, Colombo, Sri Lanka: IEEE, 2025, pp. 1–6. doi: 10.1109/ICAC69156.2025.11361528.
- [50] H. Zhang et al., "Asynchronous Threshold ECDSA With Batch Processing," *IEEE Transactions on Computational Social Systems*, vol. 11, no. 1, pp. 566–575, 2024, doi: 10.1109/TCSS.2022.3230903.
- [51] K. Tan Vo, H.-T. Nguyen, and T.-A. Nguyen-Hoang, "An Integrated Decision-Making Framework for Enhancing Blockchain Scalability and Efficiency: Layer-2, Off-Chain Storage, and Smart Contracts," *IEEE Access*, vol. 13, pp. 138328–138351, 2025, doi: 10.1109/ACCESS.2025.3596021.