LEAGAN: A Decentralized Version-Control Framework for Upgradeable Smart Contracts

Gulshan Kumar, *Senior Member, IEEE*, Rahul Saha, *Senior Member, IEEE*,
Mauro Conti, *Fellow, IEEE*, William J Buchanan, *Member, IEEE*

***Abstract*—Smart contracts are integral to decentralized systems like blockchains and enable the automation of processes through programmable conditions. However, their immutability, once deployed, poses challenges when addressing errors or bugs. Existing solutions, such as proxy contracts, facilitate upgrades while preserving application integrity. Yet, proxy contracts bring issues such as storage constraints and proxy selector clashes - along with complex inheritance management.**

This paper introduces a novel upgradeable smart contract framework with version control, named "decentraLized vErsion control and updAte manaGement in upgrAdeable smart coNtracts (LEAGAN)." LEAGAN is the first decentralized updatable smart contract framework that employs data separation with Incremental Hash (IH) and Revision Control System (RCS). It updates multiple contract versions without starting anew for each update, and reduces time complexity, and where RCS optimizes space utilization through differentiated version control. LEAGAN also introduces the first status contract in upgradeable smart contracts, and which reduces overhead while maintaining immutability. In Ethereum Virtual Machine (EVM) experiments, LEAGAN shows 40% better space utilization, 30% improved time complexity, and 25% lower gas consumption compared to state-of-the-art models. It thus stands as a promising solution for enhancing blockchain system efficiency.

***Index Terms*—Blockchain, smart contract, consensus, update, version**

.

I. Introduction

BLOCKCHAIN is one of the most promising technologies within the decentralized computing paradigm [1], with the advantages of immutability, transparency, and enhanced security [2] [3]. Overall, smart contracts in blockchain provide a programmable module stored on a blockchain [4]. These contracts run when predetermined conditions are met and where they automate the execution of an agreement. With this, all the participants in a smart contract immediately obtain the certainty of an outcome without any intermediary's involvement or time loss. Smart contracts are thus integral parts of a blockchain and attract researchers and developers to enhance performance, integrity, management alignment, coherence, and compatibility.

M. Conti is with the Department of Mathematics, University of Padua, Italy, e-mail: jmd.tannishtha@gmail.com, mauro.conti@unipd.it

R. Saha and G. Kumar are with the Department of Mathematics, University of Padua, Padua, Italy and School of Computer Science and Engineering, Lovely Professional University, India, email: rsahaaot@gmail.com, gulshan3971@gmail.com

W. J. Buchanan is with the Blockpass ID Lab at Edinburgh Napier University, United Kingdom, email: b.buchanan@napier.ac.uk

Manuscript received June XX, 2023; revised August XX, 2023.

An efficient smart contract requires some essential characteristics including being distributed, deterministic, immutable, autonomous, customizing, trustless, self-verifying, and self-enforcing. These features ensure accuracy, automation, speed-up, backup, security, multi-signature account, and information management within blockchain transactions [5]. The immutability feature of smart contracts guarantees contract performance based on the contract code and where this code is immutable and irreversible. The deployed smart contract remains the same on the blockchain forever until it is re-programmed. The history of the transactions forming part of the contract is recordable; thus, a smart contract provides strong transparency and safekeeping. On the other hand, the immutability feature of smart contracts creates a problem when an erroneous event occurs in the code, as the smart contract prohibits itself from being rectified [6]. Moreover, smart contract amendments are non-permissible even though the smart contract environmental conditions change. Overall, though the problem of immutability is relevant in post-deployment, smart contracts are mutable in the pre-deployment phase. To overcome these issues, we need improved smart contract versioning or version management [7].

Version management in smart contracts is a crucial aspect of blockchain technology that ensures decentralized applications' seamless evolution and upgradability. Smart contracts, once deployed on a blockchain, become immutable, and this makes it challenging to fix bugs or introduce new features without disrupting the entire system. This requires version management strategies. Developers often create contract versions with improved functionalities or bug fixes and can adhere to standardized upgrade patterns such as the proxy contract pattern. These new versions can be seamlessly connected to existing contracts, and preserve the integrity of the blockchain's history whilst allowing for flexible and efficient upgrades. Careful version management then not only enhances the reliability and security of smart contracts, but also fosters the growth and adaptability of blockchain-based applications in a rapidly evolving digital landscape.

A. Classification of smart contracts upgrade methods

We can classify the approaches of upgradeable smart contracts into five categories [8] [9]:

- **Contract migration**. This involves deploying a new version of the smart contract while transferring the state and data from the old contract to the new one. This offers flexibility, but requires switching to a new address.

- **Proxy-based**. This approach utilizes a proxy contract that separates storage from business logic. Overall, this allows seamless upgrades by redirecting calls to a new logic contract without changing the user-facing contract address [10]. The main drawbacks of this approach are add-on complexity and higher gas costs. Proxy patterns in smart contract upgradeability involve using a separate contract (*the proxy*) to delegate calls to another contract (*the implementation*) that contains the business logic. This allows for upgrades by changing the implementation address while keeping the same proxy address and preserving user interactions and state. For example, the proxy handles user interactions in an upgradeable voting contract while the implementation manages vote counting. If a bug is found, the implementation can be replaced without changing the proxy address, thus maintaining user trust.
- **Data separation**. This approach further enhances modularity by distinctly managing data storage and contract logic. Overall, this simplifies upgrades but introduces the challenge of coordination between multiple contracts. In contrast with the proxy-based approach, data separation methods store contract data separately from the logic. An example can be a voting contract, where a separate data contract keeps voting records and allows for logic upgrades without affecting the stored data. With a *storage contract*, a proxy calls the logic contract during code execution. Any changes are then written in the business logic part, where the original smart contract or proxy calls them through delegated calls.
- **Strategy pattern**. This enables the dynamic selection of different logic implementations at runtime through separate strategy contracts and provides flexibility for applications needing to adjust to changing conditions with added cost and complexity.
- **Diamond pattern**. This allows for multiple facets within a single contract, enabling various functionalities to coexist and be upgraded independently, but requires careful management to ensure the correct facet calls.

Each approach presents unique advantages and challenges, and which makes them suitable for different use cases in decentralized applications.

B. Motivation and contribution

The intrinsic immutability features of smart contracts restrict the post-deployment upgrade or versioning of smart contracts. However, the versioning of smart contracts is essential in the post-deployment stage due to the circumstantial changes or bug fixes. The motivation behind an upgradeable smart contract framework comes from the limitations of existing frameworks, particularly their inefficiency in managing updates and resource utilization. Traditional approaches [8] often require the redeployment of entire contracts for each update - leading to increased time complexity, higher gas consumption, and inefficient space utilization. These challenges create a research gap in developing a more efficient, decentralized system that can handle updates without compromising immutability. Therefore, in this paper, we introduce the first smart contract versioning method that combines proxy contract optimization and IPFS (InterPlanetary File System): *decentraLized vErsion control and updAte manaGement in UpgrAdeable smart coNtracts (LEAGAN)* [1]. The main contributions are:

- **Decentralized verification of update**. The verification of the changes (from an update or bug fix) in LEAGAN is a transparent mechanism that provides security to smart contract updates. Once any peer in the network initiates the update request, the peer's signature is verified. This signature verification eradicates the probability of a non-member of the blockchain initiating any update request. An update verification process follows the signature verification, where each node provides a vote for the changes. This consensus on the changes helps LEAGAN avoid any unnecessary changes.
- **Update increments**. LEAGAN allows updates in the new smart contract statements in the logic module of the existing smart contract. This process uses an incremental hash method to integrate smart contract version management and the traceability of changes. LEAGAN is the first updatable smart contract framework to use the advantages of incremental hash.
- **Revision control**. LEAGAN is the first smart contract framework to include a revision control mechanism. In this mechanism, LEAGAN compares the modification of the smart contract and adds only the changed part referring to the logic module of the data separation. This feature is advantageous in minimizing resource consumption.
- **Status contract**. LEAGAN introduces the concept of a status contract, and which applies to proxy-based and non-proxy-based upgradeable smart contracts. Overall, the storage contract or the proxy contract is immutable, and where existing implementations show the update of addresses of the logic contract or implementation contract in a storage contract. Thus, immutability becomes a contradiction for storage contracts. Our proposed solution is the first upgradeable smart contract framework to ensure the immutability of the storage contracts by using the status of all the modifications of the logic contracts separately in a status contract. This also significantly reduces the updates in the traditional upgradeable smart contracts.
- **Performance**. The incremental hash in LEAGAN shows the benefits of reduced time complexity of the smart contract updates, and revision control assists in reducing the space complexity. Thus, the combination of incremental hash and revision control systems in LEAGAN enhances the performance of our proposed smart contract framework.

C. Paper organization

Section II discusses the available approaches for smart contract updates. With Section III we outline the details

[1] In the Irish language, LEAGAN means version

TABLE I
SUMMARY OF CONTRIBUTION AND RESEARCH GAPS OF THE STATE-OF-THE-ART MODELS

Reference	Contribution	Research gap
P. Antonino et al. [11]	Proposes formal specification-based upgrades	Centralization risks due to the off-chain trusted deployer
V. C. Bui et al. [12]	Comprehensive-Data-Proxy pattern tackles re-entrancy attacks	Scalability and proxy selector clash issues
M. Rodler et al. [13]	EVMPATCH enables automated patching without source code	EVMPatch relies on bytecode rewriting, which complicates debugging and introduces compatibility issues
C. F. Torres et al. [14]	Byte-code level patching via Elysium	Fails to address modularity in upgrades
Z. Du et al. [15]	Four-tier model for smart contract upgrades	Overhead from on-chain verification
M. Salehi at al. [16]	Proxy mechanisms widely adopted	Access control concerns due to EOAs controlling proxies
T. Li et al. [17]	ATOM optimizes contract updates with differentiated code	High overhead from recompilation and redeployment
Y. Huang et al. [18]	Differentiated code for interpreting updates	Focuses on logic without considering comprehensive resource optimization
W. Shao et al. [19]	LSC for online auto-update with anomaly detection	Limited generalizability for broader blockchain applications

of our proposed LEAGAN, and including a system model following a detailed description of modular functionalities. Then, Section IV analyzes the security parameters based on existing smart contract attack models. Section V then discusses the implementation method, performance metrics, and the obtained results. Finally, we conclude our work in Section VI.

II. EXISTING SOLUTIONS

Due to their complexity and compatibility issues, smart contract updates are crucial in post-deployment. Unfortunately, existing solutions for updatable smart contracts are limited, particularly in version management. As blockchains and smart contracts are immutable, updates require altering interaction patterns rather than direct changes. In this paper, we investigate key derivations for an upgradeable framework.

Overall, smart contracts are based on the *code is law* paradigm. For this, the Ethereum community proposes auditing tools and mutable design patterns, but these only partially address the challenges of maintaining code the law effect for detecting faulty contracts and then upgrading them. Antonino *et al.* introduced a systematic framework that moves to formal specification-based smart contract upgrades [11]. This framework includes an off-chain trusted deployer to ensure the specification principle is followed. Here, the primary concern is that the deployer introduces potential centralization risks and vulnerabilities. This could compromise the decentralization principles foundational to blockchain technology.

We study a proxy-based comprehensive framework in [12]. The authors propose the "Comprehensive-Data-Proxy pattern," an innovative framework that combines data segregation with the proxy pattern to effectively defend against all types of re-entrancy attacks. The solution also addresses scalability issues associated with the proxy pattern. Experimental results demonstrate that the framework successfully tackles the challenges with minimal impact on performance. In another work, the researchers introduce an automated patching framework, called EVMPATCH [13]. It is designed to enhance the security of unmodified smart contracts without requiring source code. Utilizing a bytecode rewriter and proxy-based upgradeability, EVMPATCH enables automatic patching and redeployment in response to newly discovered vulnerabilities. It continuously runs vulnerability detection tools to ensure timely updates with minimal developer intervention. Another byte-code solution for smart contract updates is available in [14]. A study on smart contract upgradeability presents a four-tier smart contract model [15]. The model includes proxy, verification, business, and storage layers, facilitating contract linkage, integrity checks, business logic execution, and uniform data storage. Additionally, an on-chain upgrade and verification algorithm is introduced to ensure contract upgrades, verifications, and version compatibility. We observe that proxies are primarily used for upgrading smart contracts, but they are not always obvious and also raise security concerns. For example, the work in [16] evaluates six upgradeability patterns and identifies 1.4 million proxy contracts with 8,225 unique upgradeable proxies in Ethereum. The study also reveals that about 50% proxies are controlled by a single Externally Owned Address (EOA); this raises concerns about access control and governance in upgradeability. The proxy mechanism is indeed easy to implement, but proxy patterns face the challenges of smart contract compatibility, proxy selector clashing, and storage issues. We separate business logic and data storage into separate contracts in the data separation mechanism. We need a new smart contract whose address must be configured in the storage contract to update a smart contract, as this contract is immutable. Strategy patterns use satellite contracts. A new satellite contract can be developed with a new program and configured with the main contract with a new address. The diamond pattern improves the proxy pattern, where the contract can delegate function calls to multiple logic contracts.

In a recent work [17], the authors show a new smart contract architecture and optimization mechanism, called ATOM, which provides architectural support to update contracts economically and fast executing instruction-wise. Generally, the smart contract architecture and updating mechanisms are low speed and cause high overhead due to the recompilation and redeployment process. ATOM addresses these issues and shows its efficiency in updating the smart contract. The use of differentiated code for the updatable smart contracts is noteworthy [18]. Differentiated code is the source code except for the repeated ones in two similar smart contracts. This shows how a software feature is implemented or a programming issue is solved. Such differentiated code is used to interpret the update of a smart contract in its next version. LSC, a framework for online auto-update smart contracts in blockchain-based log systems is shown in [19]. LSC is a secure updatable smart contract mechanism with time-varying log anomaly detection. It uses self-adaptive machine learning

log anomaly analysis and is continuously fed to the contracts. LSC uses shared anomaly detection strategies for audit log systems to defend against targeted detection evasion. Table I outlines the contribution and research gap for each of the above-mentioned works.

The literature survey highlights significant challenges in upgradeable smart contracts, including centralization risks, proxy selector clashes, bytecode rewriting complexities, scalability issues, access control vulnerabilities, and inefficient resource utilization. Existing frameworks rely heavily on proxy-based mechanisms, differentiated code approaches, and auto-update models. However, these models show limitations such as high gas consumption, complex debugging, storage inefficiencies, and lack of version traceability. Many solutions require off-chain trusted deployers, which leads to centralization risks or inefficient recompilation processes and increases computational overhead. Addressing these gaps requires a decentralized, efficient, and modular framework that can manage contract updates, optimize resource utilization, and ensure security. Our proposed LEAGAN provides a solution to the above-mentioned problems integrating incremental hashing, revision control, and decentralized verification; this enables efficient upgrades, transparent governance, and immutability, reducing computational complexity while enhancing security, scalability, and flexibility in blockchain-based upgradeable smart contract management.

III. Proposed framework: LEAGAN

LEAGAN is the first comprehensive and decentralized smart contract framework to include incremental hash-based updates, revision management, and update verification. In this section, we first describe the system model (Section III-A) of our proposed LEAGAN followed by the module-wise functionalities (Section III-B). Note that we discuss only the change and update in a smart contract assuming all the other basic functionalities of a smart contract are the same. We add Table II to summarize the notations used to discuss our proposed LEAGAN functionalities.

TABLE II
Summary of important symbols and their definitions

Symbol	Definition
Req_{c_u}	Change request by user u
Sig_R	Signature of a requester u
e	Error
t	Timestamp
M	A message
M'	A modified message
H	Hash family
$HGen$	Probabilistic Polynomial Time (PPT) generator
$HEval$	Polynomial time hash evaluation
$H(M)$	Synonymous to $HEval$
$IncH$	Incremental hash function
h	Calculated hash value
k, b, n	Integers to represent the powers
m	The changes in a message
$\mathcal{C}$	Collision-free compression function
i	index value for a series
K	Keccak-256 construction
$\mathcal{S}$	Original contract statement
$\mathcal{S}'$	Updated or changed statement to be included in the logic contract
$Addr_{CL'}$	Byte code address of updated contract

A. System model

In Figure 1, we show a comprehensive system model for our proposed LEAGAN. The blockchain consists of a peer-to-peer network. Any peer has the right to notify any legitimate error or bug in the smart contract. Besides, we also open the scope of the upgradeable smart contract not for error or bug fixes, but LEAGAN also allows to include a verified change in the form of any program condition or statement in the smart contract. Thus, LEAGAN becomes a novel generic framework for blockchain upgradeable smart contracts. Any peer (change requester) requesting a change broadcasts a change request message in the blockchain network. The other peers in the blockchain network verify the signature of the requester and the change request. We segregate the smart contract composition into three parts: business, data, and status. We can also add the term contract to each of them to maintain the notion of automated transactions. In an upgradable smart contract scenario, the logic contract contains the business logic (mutable), the data storage holds the contract's state and data (immutable), and the status contract tracks the version and update status (mutable) of the logic contract. When an upgrade occurs, the logic contract is updated while the data storage remains unchanged, and the status contract records the new version; this ensures continuity and transparency, i.e., non-halt deployment. Once the signature and the change required parameters are verified, LEAGAN uses an incremental hash to update the business logic in the smart contract. At the same time, LEAGAN uses a revision control system to manage the versions of the smart contract and updates the address of the business logic in the data storage. This mechanism is integrated directly into the LEAGAN framework, functioning as part of the smart contract system rather than as a separate application. It is designed to seamlessly operate within the blockchain environment, ensuring that updates are decentralized and transparent. We discuss all the details of the above-mentioned functionalities in Section III-B.

B. Functionalities

In this paper, we assume that the basic functionalities of a smart contract are the same as those of our proposed LEAGAN. Therefore, we discuss only those functionalities, which denote LEAGAN to be novel and efficient for blockchains. The functionalities include change request verification, incremental hashed updates, version management through a revision control system, and non-halt redeployment.

1) Change request verification: A peer u in a blockchain notifies an error or bug by broadcasting a change request Req_{c_u} that contains the signature Sig_R of the requester u, the error e, the required change c, and the time stamp t. Once Req_{c_u} is received by the peers in the network, the peers verify the signature by using the public key available in the blockchain network. After the signature verification, the other request parameter, such as timestamp, error, and the required change, is verified. Timestamp verification helps avoid a replay

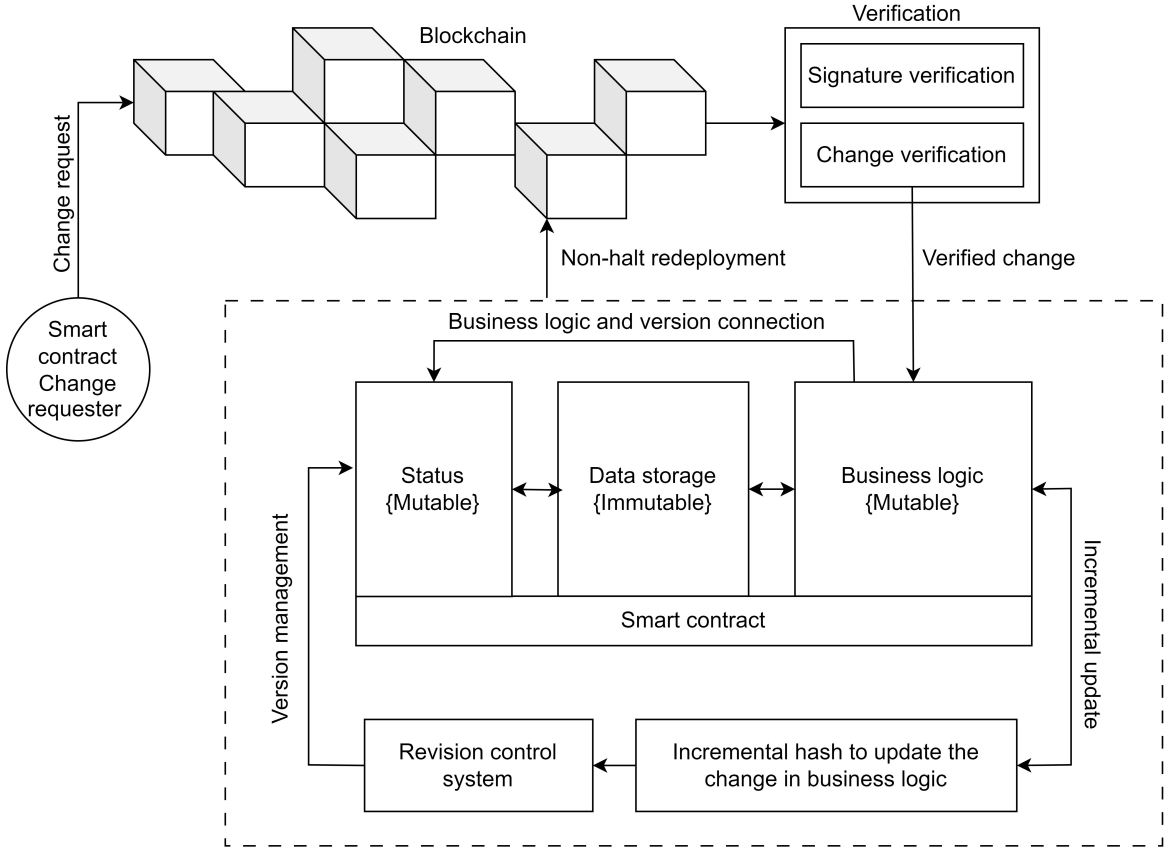


Fig. 1. System model for proposed LEAGAN framework

of the request; error verification and change verification help in the transparency of the smart contract update. We use i to denote the index of the number of users or peers requesting a smart contract change, and where j is an indexing parameter used for verification of the signature in the change request. We need to compare the index values in order to avoid the problem of self-validation of the signature. The maximum value of u_i can be the number of peers online and requesting the requests at a time. We summarize the process in Algorithm 1.

2) Incremental hashed updates: Incremental hash functions are commonly used to update a hash value by changing a message from M to M'. These are calculated from the existing hash value of M and without starting the calculation from scratch. Thus, in contrast to conventional hash functions, incremental hashes are faster and more resource-efficient. In LEAGAN, we apply incremental hash for memory checker. Here, we first describe the fundamentals of incremental hash and then show the application of incremental hash in LEAGAN.

a) Fundamentals of incremental hash: Incremental hash uses two functions: $HGen$ and $HEval$ as shown in [20]. Following the definition in [20], H is a family of hash functions consisting of two functions $(HGen, HEval)$. $HGen$ is a Probabilistic Polynomial Time (PPT) generator that takes inputs of $1^k, 1^b, and 1^n$ and outputs a string H. $HEval$ is a polynomial time hash evaluation algorithm that takes H and a message $M \in {B_b}^n$ as inputs and outputs a k bit string called the hash of M under H. In particular, $H(M)$ and $HEval(H, M)$ are synonymous. Incremental hash uses an update algorithm $IncH$ for H with running time $T(,,)$ if $\forall k, b, n, \forall H \in [HGen(1^k, 1^b, 1^n)], \forall j \in$

Algorithm 1 Verification process in proposed LEAGAN

Input: $Req_c : Sig_R, t, e, c$
Output: verified Req_{cu}

$u_i : \overrightarrow{broadcast}\ Req_{cu} : Sig_R, t, e, c$
for $j = 0$ to $n - 1$ **do**
  $u_j \neq u_i$: verify (Sig_R)
  if verify $(Sig_R) == TRUE$ **then**
    verify(t, e, c)
    if verify $(t, e, c) == TRUE$ **then**
      Return Req_{cu} verified
    else
      Abort Req_{cu}
    end if
  else
    Abort Req_{cu}
  end if
end for

$1, 2, ..., n, and\ \forall\ m \in B_b$. If $h = HEval(H, M)$, then it is the case that $IncH(H, M, h, (j, m))$ halts in $T(k, b, n)$ steps with an output equal to $HEval(H, M < j, m >)$. The proof of collision-freeness of this incremental hash and its security analysis is available in [20].

b) Incremental hash definition for LEAGAN: Our method uses the Chaining-based Incremental Hash Function (PCIHF) as defined in [21].

- $HGen$: We use $\mathcal{C}$ for a collision-free compression function with n-bits input and a k-bits random output string. Using $\mathcal{C}$, we obtain a series of intermediate hash values for $i \in [1, n-1]$, so that the following equation holds:

$$h[i] = \mathcal{C}(m[i]||m[i+1]), \tag{1}$$

  where $m[i]$ is a bit in the input message to $\mathcal{C}$. To ensure the overall system's security and efficiency, a modular addition operation is beneficial as a combining operator. This is associative, commutative, and invertible in a particular group. The hash function is also parallel and incremental. We fix the length of the final hash value H to L in the generalization. We can calculate the final hash value as follows:

$$H = \sum h[i](mod\ 2^k + 1), \tag{2}$$

  where $i \in [1, n-1]$.
  Though the above generalization applies to $HGen$, it is applicable to all hash functions. For the simplicity of the implementation, we follow the hash functions for Ethereum Virtual Machine (EVM), i.e., Keccak-256 construction (K).
- $HEval$: $HEval$ takes three inputs: the hash function $h[i]$, the target message M_t, and the data to be updated M_u.

Algorithm 2 Example of logic contract updates

```
contract ContractLogic1 is
    state variable: uint value
    function initialize()
        begin
            value ← 0
        end function
    function updateValue(uint x)
        begin
            value ← value + x
        end function
end contract
Modified logic contract example
contract ContractLogic2 is
    inherits: ContractLogic1
    function updateValue(uint x)
        begin
            value ← value + x + 20
        end function
end contract
```

c) Incremental hash in LEAGAN: As stated earlier, LEAGAN follows a data separation method for smart contract updates. Thus, the LEAGAN framework separates the smart contract into two parts: a business logic contract or logic contract and a data storage contract or storage contract. The logic contract address is stored in the storage contract, and the storage contract must be configured with the logic contract address at the time of deployment.

LEAGAN exploits the utility of the data separation method and shows that the configured address in the storage contract is updatable as required with the help of the incremental hash. Note that the storage contract update or upgrade means a change in the memory address of the logic contract. For this, we create two logic contracts to experiment with LEAGAN: $ContractLogic1$ and $ContractLogic2$ - as shown in Algorithm 2. Within the implementation, $ContractLogic1$ is the first to be implemented, whereas $ContractLogic2$ contains an update from $ContractLogic1$, that is, $value+ = 20$. As per the traditional approach, each logic contract should have separate deployment addresses in the blockchain of $Addr_{CL}$ and $Addr_{CL'}$, respectively. These addresses are calculated using the deployer address (sender) and the number of transactions this account sends (nonce). The deployer address and the nonce are Recursive Length Prefix (RLP)-encoded [23] and hashed with Keccak-256 [22]. Thus, calculating the hashed value (address) for each update in the logic contract uses the same approach and starts from scratch. It is admissible for different smart contract deployers; however, for the same deployer and the same smart contract, the classical approach of address generation consumes a significant amount of time and resources, leading to more overhead in the blockchains. Therefore, LEAGAN uses incremental hash for the updates in the smart contracts from the same deployer to reduce the time and space complexity. We summarize the process in Algorithm 3, and where we assume that the original statement in the contract is $\mathcal{S}$ and the updated or changed statement to be included in the logic contract is $\mathcal{S}'$. To obtain the deployment address of the changed or upgraded smart contract ($Addr_{CL'}$), we calculate the bytecodes of $\mathcal{S}$ and $\mathcal{S}'$. Along with this, we calculate the intermediate hash value y. Finally, we use $HEval$ to calculate the deployment address, using y, $Addr_{CL}$, and $\mathcal{B}'$.

Algorithm 3 Incremental hash in LEAGAN

Input: $\mathcal{S}, \mathcal{S}'$
Output: $Addr_{CL}*$

$y = h_1 \boxplus_{256} h_2 \boxplus_{256} ... \boxplus_{256} h_n$
$\mathcal{B} = bytecode(S)$
$\mathcal{B}' = bytecode(S')$
$Addr_{CL} = K(RLP(address||nonce)$
$h = HEval(y, Addr_{CL}, \mathcal{B}')$
$H = y \boxplus_{256} h$
$Addr_{CL'} = H$
Return $Addr_{CL'}$

Address update: In our LEAGAN framework, each update to the smart contract, whether it is an addition, modification, or deletion, produces a new hash. This new hash is then incrementally combined with the previous one, representing

the prior version of the contract. This process creates a unique identifier for the current version of the contract. The hashed value is then used to determine the address of the updated contract on the blockchain. As a result, each version of the contract has a distinct address derived from its unique incremental hash. This ensures the traceability and immutability of the validated changes.

It should be noted that the process of calculating the incremental hash and determining the address of the current version of the smart contract is decentralized and performed by all nodes in the blockchain network. Each node then independently verifies the correctness of the incremental hash and the resulting contract address during the validation of transactions. This process ensures all nodes reach a consensus on the contract's current state and corresponding address. The decentralized nature of this process enhances the security and integrity of the smart contract execution on a blockchain as no single entity or separate application has control over the update process. Instead, it relies on the collective agreement of all participating nodes in the network.

3) Version management: The use of incremental hash assists in managing versions and their traceability is a transparent way in the blockchain framework. Each incremental hash is a version of the originally deployed smart contract. We then apply the concept of Revision Control System (RCS) in version management for resource utilization [24]. This is illustrated in Figure-2 that shows that the storage contract and logic contract $V_{1.1}$ (a generalized smart contract combining these two parts) -and which are deployed in the blockchain address. A change or modification or upgrade occurs in $V_{1.1}$ leads to logic contract $V_{1.2}$ to $V_{1.n}$. However, to update the logic contract address, we only use the modified parts for an incremental hash calculation to resolve the new smart contract address. This helps to reduce the overhead and consumption of time and resources.

a) Version dimensionality: The literature shows that the existing smart contract upgradeable frameworks are one-dimensional. It means that once a smart contract is upgraded to a new version, it cannot be reverted to a previous version. This one-directional nature implies that the upgrade process is linear - and when changes are made and the contract is deployed, the earlier version becomes inaccessible. This can pose challenges for developers and users, and any issues or bugs introduced in the new version may require the previous versions to work around or to provide a reference. LEAGAN solves this using the RCS, and where every version is incrementally hashed to the upgraded version. As a result, the history of upgrades is accessible on-chain.

b) Storage space utilization: In LEAGAN, the integration of the RCS allows for efficient storage space utilization through the concept of incremental hashing. When a smart contract, such as logic contract V1.1, needs to be upgraded, RCS focuses on identifying and storing only the modified portions of the contract, rather than redeploying the entire contract. For example, if V1.1 is modified to create V1.2, RCS calculates an incremental hash based on the changes made. This hash serves as a version identifier, allowing the new contract version to reference the original contract, while incorporating only the differences. This method significantly reduces the amount of data stored on the blockchain as it avoids redundancy. By using this technique, LEAGAN minimizes storage requirements and bandwidth consumption, as only the changed segments are logged. Additionally, developers can efficiently track versions as the blockchain maintains a transparent history of changes through incremental hashes. This leads to a more resource-efficient blockchain environment and enables better scalability while maintaining the integrity of the contract versions.

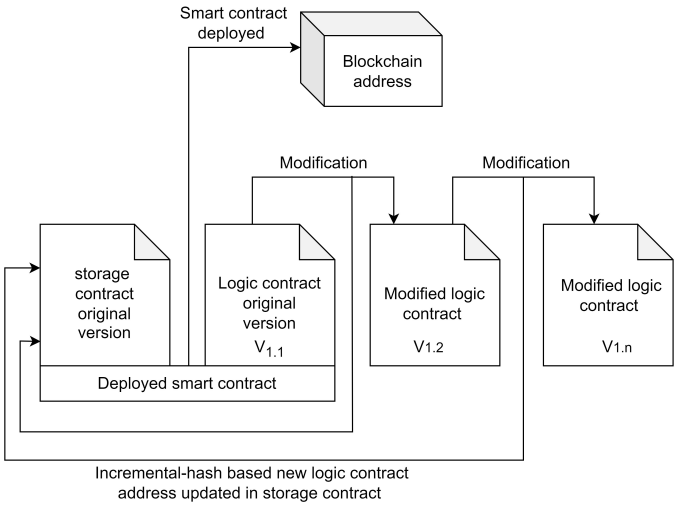


Fig. 2. RCS in proposed LEAGAN

4) Non-halt deployment: Generally, it is believed that a smart contract cannot be changed once deployed. Existing research identifies that the proxy contract concept involves an a changeable implementation contract (business logic), whereas the proxy contract (address and storage) is non-changeable. However, we observe in the existing proxy-based smart contracts that the memory address of the new implementation contract must be updated in the proxy contract. This means that a proxy contract is also changeable, which contradicts the claim of the immutability of proxy contracts or storage contracts. In LEAGAN, we first introduce the concept of a status contract, which contains the addresses (changeable) of all the logic contracts. We extend Figure 2 to show the concept of status contracts in Figure 3. In this figure, we show how the immutability of the storage contract is maintained in the presence of our new status contract. A user then deploys a storage contract with its address $uint\ value$ and the address of the status contract $addr_statcontr$. When the storage contract is invoked with a triggered condition, it calls the $StatusContract()$. The status contract contains $uint\ value$ that points to the address of the storage contract, status contract code (conditions for executing various logic contracts based upon their modification status, and addresses of the logic contract versions. The dashed arrows from the status contract to the logic contract versions denote the respective pointer.

For the conditions of the status contract, we use two values of 0 and 1. We consider 0 as the last modified version of a logic

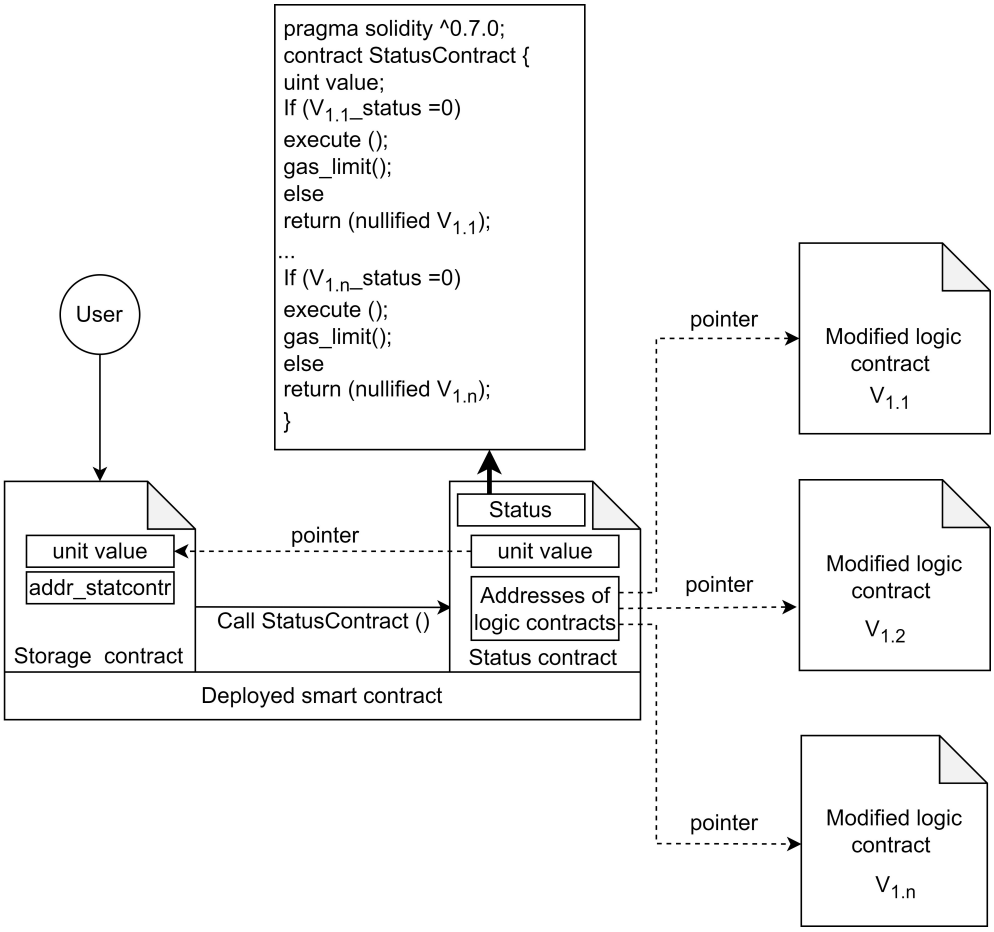


Fig. 3. Status contract in proposed LEAGAN

contract, whereas 1 represents the changes already made. As all the versions' addresses are available in the status contract, it is easy for traceability and accountability. Thus, the feature of a status contract adds novelty to LEAGAN. Moreover, this status contract helps maintain the immutability of the storage contract (proxy contract in proxy-based methods) and avoids any halts in the smart contract execution. We include a novel feature in the status contract, i.e., the gas limit function for sub calls $gas\_limit()$. A gas limit denotes the threshold value of gas required for a smart contract transaction execution. This helps to balance the supply and demand of the gas in the network.

IV. Security analysis

In this section, we discuss the security features within LEAGAN, first discussing the attack circumstances and providing the security notions of LEAGAN against attacks.

A. Attack circumstances

Attack circumstances (AC) are the situations or events in a threat model, where an adversary $\mathcal{A}$ attempts to manipulate the system's normal functioning. We consider the following attack circumstances to prove the security strength and the resistivity of our proposed LEAGAN:

- ***AC1***: In general upgradeable smart contracts consider the storage contract to be immutable. When we develop a new version of a logic contract, the new deployment address needs to be updated in the storage contract. An adversary $\mathcal{A}$ is then able to exploit this feature to include a malicious address $(Addr^{M}_{CL'})$ of the new logic contract CL', which, in turn, executes a malicious program for the logic contract. Thus, smart contract outputs wrong transfers. Mathematically, we can provide the following interpretation for this AC as:

$$\mathcal{A} : deploy(CL') = \mathcal{M}(Addr_{CL'}) = Addr^{\mathcal{M}}_{CL'}, \quad (3)$$

where $deploy(CL')$ is the deployed address and $\mathcal{M}$ is malicious function run by $\mathcal{A}$ to generate the malicious address.
- ***AC2***: The re-entry attack is one of the most concerning attacks in smart contracts, and where storage contract generally calls to the logic contracts (externals). An adversary $\mathcal{A}$ exploits the external calls and forces the contract to execute and call back itself using a fall-back function. Thus, executing the code *reenters* the contract leading to recursive calls. Following AC1, the attacker can carefully construct a contract at an external address that contains malicious code in the fallback function and

attempts to drain funds from the smart contract using the withdraw function.
- ***AC3***: A smart contract execution on the Ethereum blockchain platform requires a gas fee - and which is used for the incentives for validators (miners). The price of gas is time-based and is determined by supply, demand, and network capacity. Gas griefing is an attack where a user sends the amount of gas required to execute the target smart contract, but not enough to execute subcalls.
- ***AC4***: The distributed characteristics of the Ethereum platform make it almost impossible to ensure a synchronous time on every node. An adversary $\mathcal{A}$ uses asynchronous timestamp values to craft a logical attack against any logic contract. This problem further leads to stale contract calls.
- ***AC5***: An adversary $\mathcal{A}$ attempts to search for a collision for the Keccak algorithm as it is the core component used in our incremental hash (PCIHF).

B. Security solutions by LEAGAN

In this part, we discuss the security solutions provided by LEAGAN to mitigate the previously mentioned ACs:

- ***Solution for AC1***: LEAGAN uses version management without deleting the old logic contract address; rather, all the new versions of the logic contract are stored in the status contract. This new concept makes the storage contract separate and perfectly immutable. We also use 0 and 1 to avoid redundancy among the upgradeable logic contracts' versions and mitigate the problem of stale - invalidated due to versioning - logic contract calls. All the processes of status contracts in LEAGAN are transparent, and thus, avoids any memory manipulation problems.
- ***Solution for AC2***: LEAGAN updates the contract's state in the proposed concept of status contract rather than the traditional storage contract. A further time-locking mechanism is applicable. We keep this discussion for future scope. However, we can use Slither, Mythril, and Securify to check for the presence of the different types of re-entrancy vulnerabilities.
- ***Solution for AC3***: LEAGAN proposes to use a gas limit in the status contract. When a smart contract is called LEAGAN executes the storage contract that calls the status contract. In the status contract for each sub-call, we provide a gas limit. This helps miners decide on the execution of the transaction validation without any ambiguity regarding the gas fees for calls and sub-calls. Thus, we can mitigate the problem of gas griefing and keep the detailed discussion on $gas\_limit()$ for future scope.
- ***Solution for AC4***: LEAGAN controls the version management using the flag values of 0 and 1, rather than using the timestamps. Thus, timestamp-based calling to a contract or subcontract is not allowed in LEAGAN, and thus, timestamp-based attacks are eradicated.
- ***Solution for AC5***: Existing literature shows that PCIHF is not only universal one-way but also collision-free [21]. We choose Keccak-256 as the randomizer and the modular addition operator as the combining operator. Keccak instances with a capacity of 256 bits offer a generic security strength level of 128 bits against all generic attacks, including multi-target attacks. Keccak's security proof allows an adjustable level of security based on a *capacity* (c) and providing $c/2$-bit resistance to both collision and pre-image attacks [22].

The above discussion shows that our proposed LEAGAN is efficient in terms of security and provides a solution to some critical attacks in smart contracts.

V. Performance evaluation

In this section, we first describe the implementation strategy for LEAGAN, including the network model and the required technical attributes for reusability. We also define the evaluation metrics important for measuring performance, followed by the detailed results and comparative analysis.

A. Implementation

a) Hardware and software requirements: To analyze the practicality of LEAGAN, we conduct the experiments on a system with the following specifications: an Intel Core i7-12700K processor with 12 cores (eight performance and four efficiency cores) and 20 threads, operating at a base clock speed of 3.6 GHz with a boost clock of 5.0 GHz. The system had 16 GB of DDR4 RAM running at 3,200 MHz and a 1 TB NVMe SSD for high-speed storage. The operating system used was Ubuntu 22.04 LTS (64-bit). The experiments use a local Ethereum test network (Ganache CLI) to simulate real-world blockchain interactions. These specifications were chosen to ensure a balanced evaluation of LEAGAN's resource efficiency under realistic and modern computing environments.

b) Proposed framework requirements: We use Ethereum to observe LEAGAN's behavior, evaluate its performance, and show the implemented network model in Figure 4. This includes essential components for implementing LEAGAN in the implementation process. These include Solidity, Web3.js, Infura.io, Ethereum Networks, Cloudflare's Ethereum Gateway, Ganache CLI, Solc, and Metamask.

Figure 4 illustrates the implementation flow of LEAGAN on Ethereum. The process begins with smart contracts written in Solidity and compiled into bytecode using Solc. This bytecode is then deployed on the Ethereum blockchain via Web3.js, and interfacing with Infura.io or Ganache CLI for network connectivity. During execution, Metamask acts as a wallet for managing transactions. The Ethereum Virtual Machine (EVM) ensures uniform contract logic execution across nodes. LEAGAN incorporates a status contract for version management and uses incremental hashing to optimize updates. This setup integrates tools like Cloudflare's Ethereum Gateway for secure communication and ensures a seamless and efficient contract execution pipeline. In Table III, we summarize the functioning of the above-mentioned components in implementing LEAGAN.

c) Methodology and deployment: The implementation of LEAGAN on Ethereum involves a coordinated interaction among the components mentioned in Table III:

- Solidity is used to write the smart contracts, which are then compiled into bytecode using Solc (Solidity Compiler). We deploy the bytecode on the Ethereum network via Web3.js; it is a JavaScript library that facilitates interaction between the client-side application and the Ethereum blockchain.
- Ethereum Virtual Machine (EVM) ensures that the smart contract logic runs consistently across all Ethereum nodes, making LEAGAN's behaviour deterministic and secure across the network.
- Infura.io provides a scalable, remote connection to Ethereum nodes and enables LEAGAN to interact with the Ethereum network without running a full node.
- Cloudflare's Ethereum Gateway ensures secure and reliable communication with the Ethereum blockchain, acting as an intermediary that optimizes data transfer.
- Ganache CLI creates a local Ethereum environment for testing and debugging before deploying contracts on the mainnet and simulates the blockchain, allowing developers to perform tests in a controlled environment.
- Metamask serves as the user's Ethereum wallet, enabling transaction management and interaction with the deployed LEAGAN contracts.
- JSON-RPC is a Remote Procedure Call (RPC) protocol used to interact with the Ethereum network. JSON-RPC enables communication between the client-side application (using Web3.js) and the Ethereum node, such as those provided by Infura.io or Ganache CLI. Through JSON-RPC, Web3.js sends requests to the Ethereum node to perform operations such as deploying contracts, querying blockchain data, and sending transactions.

TABLE III
TOOLS USED IN THE EXPERIMENTATION AND THEIR PURPOSE

Tool used	Purpose
Solidity	Smart contract programming language for Ethereum.
Web3.js	It is a JavaScript library. It enables web browsers and Node.js to read and write to the Ethereum blockchain. Solidity-based smart contracts are executable using JSON RPC from Web3.js.
Infura.io	It is an API for simple Ethereum network access through HTTP and WebSockets.
Ethereum Networks	The Ethereum Main Network (a.k.a Homestead) is considered to be the production environment.
Cloudflare's Ethereum Gateway	It is a free API for accessing the Ethereum Main Net.
Ganache CLI	It is a command line interface for running a locally hosted instance of Ethereum. We can start up a blank Ethereum blockchain, or a fork of a public network's blockchain on your local machine.
Solc	It is a Solidity compiler
Metamask	This facilitates the invocation of Ethereum smart contracts from a web page.

B. Evaluation metrics

To evaluate the performance of LEAGAN, we consider some important attributes: Transactions per second (TPS), gas consumption, contract decision and update accuracy, and resource utilization.

- ***Throughput***: We define throughput based on the number of transactions successfully written on the blockchain. Generally, Ethereum provides 10-12 transactions per second (TPS).
- ***Gas consumption***: Gas is the computational effort required to execute specific operations on the Ethereum network. Each Ethereum transaction requires computational resources to execute; thus, each transaction requires a fee to utilize the resource. We use *gasleft()* function twice (once at the beginning of the transaction $startGas$ and once at the end of the transaction) in solidity for the measurement of gas consumption. The mathematical expression is given below. We use Equation 4 at the beginning to measure the initial gas and Equation 5 at the end to measure the overall consumption.

$$startGas = gasleft() \tag{4}$$

$$gasused = startGas - gasleft(); \tag{5}$$

We measure the gas consumption concerning the number of blocks varying from 10 to 100, assuming each block has an equal number of transactions. This involves measuring the average gas consumption of all the frameworks under consideration. Note that we specifically calculate the gas consumption for operations related to contract updates. The gas consumption measurement includes the transaction fees for the main calls and the associated sub-calls involved in updating the smart contracts. The experimental results for gas consumption thus focus on the operations directly related to contract updates, such as deploying new contract versions, interacting with the status contract, and managing incremental updates.

- ***Contract decision and update accuracy***: We introduce the contract decision and update accuracy Acc feature to measure the accuracy of the updates in the upgradeable smart contracts. We measure this metric using:

$$Acc = \frac{Acc_u}{N_u} \times \frac{Acc_d}{N_u}, \tag{6}$$

where Acc_u represents the number of accurate updates, N_u denotes the total number of updates, and Acc_d denotes the number of accurate decisions from the smart contract in the presence of updates.

In the context of smart contract frameworks, an accurate update refers to a scenario where the updated contract logic is correctly executed without causing transaction failures, halts, or incomplete transactions. For example, if a decentralized finance contract is upgraded to include a new feature for calculating interest, an accurate update would ensure that all relevant sub-calls (like checking user balances, applying interest rates, and updating records) are correctly executed without errors and the final state reflects the intended logic. On the other hand, an inaccurate update would lead to issues such as failed transactions or incorrect outcomes. For instance, if the same decentralized finance contract update overlooks a critical sub-call that verifies user eligibility for interest, it could lead to inaccurate results, such as applying

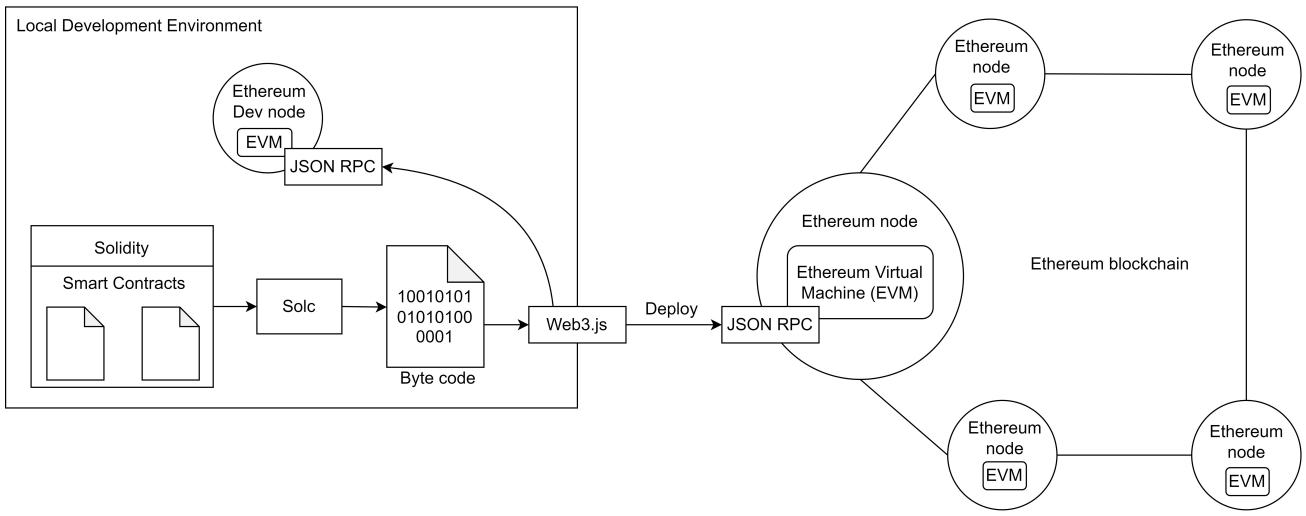


Fig. 4. Implementation model for LEAGAN

interest to ineligible accounts. This causes the system to make misleading decisions, such as incorrectly reporting account balances or failing to execute subsequent transactions.

- ***Complexity analysis***: We analyze the overall algorithm and compute the time complexity and space complexity to show and validate the enhanced features of our proposed LEAGAN.
- ***Resource utilization***: We measure resources in three dimensions: CPU, storage, and disk utilization. CPU utilization measures the percentage of processing power the system uses during the execution of smart contract update operations. This can be mathematically measured by programming the scripts to initialize at an invoke of an update operation till the complete smart contract has been updated. It evaluates the computational efficiency of the proposed framework. Disk Input-Output (I/O) refers to the percentage of reading and writing operations required during smart contract update operations; this metric is essential to understand the I/O overhead introduced by the framework. Storage utilization evaluates the amount of blockchain storage space consumed by smart contract update operations. Efficient storage utilization indicates reduced redundancy and better scalability.

C. Experimental results

We have mentioned earlier that LEAGAN is the first upgradeable/mutable smart contract framework to exploit the potentials of incremental hash, a revision control system, a novel status contract, and decentralized verification. The existing literature shows some frameworks in the direction of upgradeable smart contracts; however, most frameworks focus on proxies. Therefore, we choose three existing concepts of upgradeable smart contracts: i) Proxy-based [10], ii) Differentiated code-based [18], and iii) Auto-update based [19]. The proxy-based approach uses a proxy contract to separate logic and storage, enabling upgrades by redirecting calls to new logic contracts. However, it suffers from proxy selector clashes, storage issues, and access control vulnerabilities, making it a widely used but flawed solution. The differentiated code approach interprets updates by storing only the modified logic, reducing redeployment overhead. While efficient, it neglects storage optimization and traceability. The auto-update approach leverages machine learning for anomaly detection and secure updates, but its narrow focus on logging systems and high computational overhead limit scalability. These baselines highlight LEAGAN's ability to address storage, traceability, and resource optimization comprehensively. Though the concepts of internal methodologies differ among the selected state-of-the-art models, we evaluate the algorithms using the same parameter settings as our proposed LEAGAN. Throughout the experiments, we have executed overall 200 updates with a chain of 100 blocks; this gives an update rate of 2. The updates include access control modifications, logic refinement, external contract integration, bug fixing, and exception handling. Note that an average update rate of 2 means that two update transactions occur per block; it does not imply that each block exclusively contains only update transactions. Other transactions, such as token transfers, contract interactions, and data read/writes, also exist in the blocks, which are characterized by typical blockchain activities, including financial transfers, state updates, or calls to contract functions.

1) Throughput evaluation: We know that blockchains face scalability issues. Therefore, we compare the throughput based on the smart contracts in the blockchain frameworks. We process overall 100 blocks and compare the average throughput for all the models. We show the obtained results in Figure 5.

From the figure, we observe that our proposed LEAGAN shows stable outputs of throughput with an average TPS value of 23.36. The other models of [10], [18], [19] show average TPS outputs of 17.37, 17.69, and 14.94, respectively. This shows that the proposed LEAGAN is 27.3% better in terms of throughput on average. We obtained this enhanced throughput by using incremental hash, which has low complexity and faster computation. Besides, our new smart contract configuration helps us achieve such throughput.

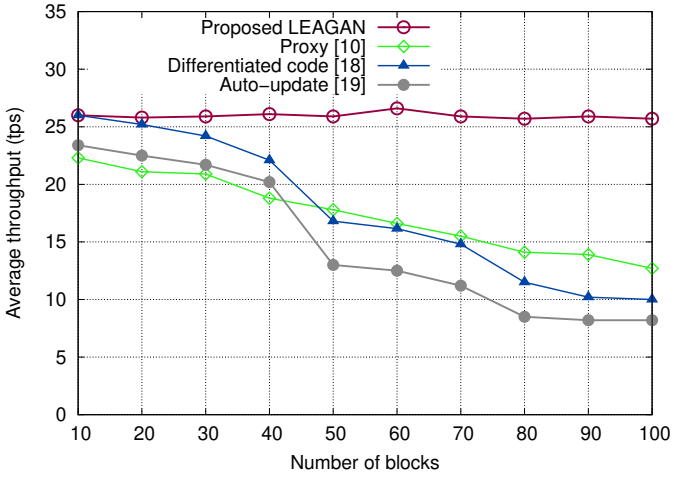


Fig. 5. Throughput comparison

2) Gas consumption evaluation: We show the gas consumption results of our proposed LEAGAN and other frameworks in comparison in Figure 6. The average gas consumption values for [10], [18], and [19] are 27452, 25963, and 32278, respectively. In LEAGAN, the measured gas consumption averages $\sim 22,947$ gas per transaction, which is 24.5% better as compared to the existing frameworks in gas consumption on average. This reduction is achieved by optimizing token transfers, contract calls, and subcalls. ERC-20 token transfers, involving balance updates in storage, typically consume $\sim 41,000 - 46,000$ gas in conventional models. However, LEAGAN minimizes these costs by batch-processing updates and reducing redundant state modifications. Additionally, contract interactions, particularly calls and delegate calls, are optimized through the status contract mechanism, eliminating unnecessary storage writes. Calls within LEAGAN, including external contract interactions (CALL opcode) and state modifications ($SSTORE$), consume $2,600$–$7,000$ gas per operation. Given the $40,000$ gas limit, LEAGAN operates within a safe execution margin ($\sim$ $22,947$ gas), preventing transaction failures while maximizing efficiency. From Figure 6, we observe that the existing frameworks tend to move towards higher gas consumption, eventually leading to failure.

3) Contract decision and update accuracy: The existing frameworks do not consider the sub-calls in the smart contracts; this leads to unnecessary transaction halt, incomplete transactions, and even transaction failure. We show the accuracy of the contracts (after updates) in Figure 7. In LEAGAN, accurate updates are ensured by decentralized verification and

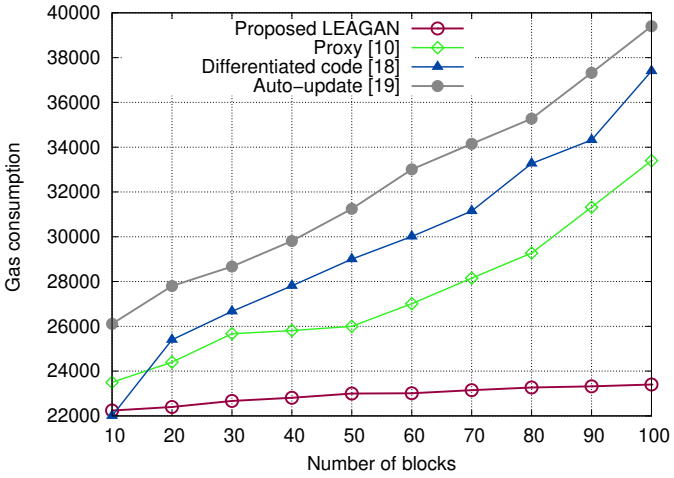


Fig. 6. Gas consumption comparison

tracking through a status contract, which uses flag values (0 or 1) to denote the freshness of the contract versions. This mechanism helps LEAGAN to maintain 100% accuracy by ensuring that only valid and verified logic is used; thus, LEAGAN prevents the problem of incomplete or inaccurate updates observed in other frameworks. Moreover, we can see from the figure that the existing frameworks' performance for accurate updates degrades with time due to their inability to effectively manage sub-calls and track contract updates over time. As the blockchain grows, outdated or unverified logic (if not managed) can lead to incorrect executions, causing transaction failures, incomplete operations, and misleading decisions, reducing overall accuracy. The figure shows that the proposed LEAGAN efficiently provides 100% accuracy with increasing block numbers. The upgrade or update requested is decentralized verified and separately stored in the status contract. The status contract further uses flag values of 0 or 1 to denote the freshness of versions of logic contract references stored in the status contract. Thus, the accuracy of LEAGAN is optimum. The other frameworks cannot achieve 100% accuracy (marked in green color on the upper limit of the y-axis); instead, the frameworks start to make misleading decisions based on the updated smart contract conditions or parameters.

4) Complexity and resource utilization analysis: We measure the complexity in two aspects: time and space. Blockchains are very complex; therefore, any add-on complexity may lead to the system's failure. We consider a smart contract update function $\mathcal{U} = replace(m, \delta_h)$, where m is the modified data, and δ_h denotes the new hashes of the smart contract in the blockchain after the update. Therefore, the new hash h' is connected to a smart chain becomes as:

$$h' = h + R(S[i]||S[j]) + \mathcal{C}_n(\delta_h), \tag{7}$$

where $S[i]$ is the target logic contract for modification, $S[j]$ represents subsequent modifications in the logic contract and updated in the status contract, and R represents the replace function. The subsequent blocks for $j = 0, 1, ..., n-1$, $\mathcal{C}$ is

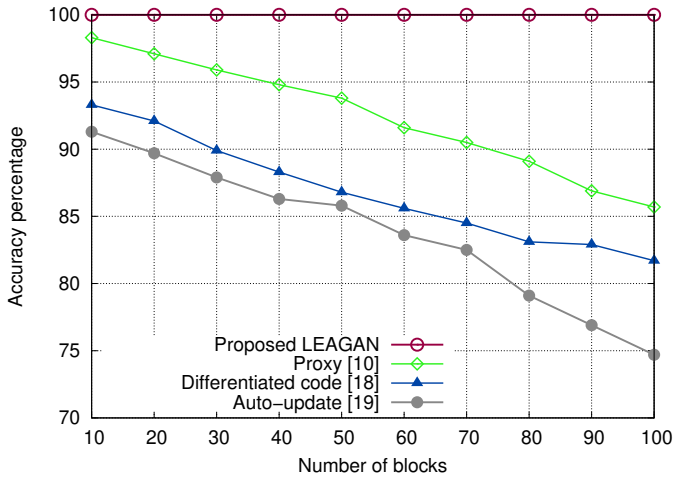


Fig. 7. Accuracy of smart contract updates

the combiner function, and n is the total number of blocks in the blockchain. We calculate the total cost as:

$$Cost_T = 2(1+p)(R_t + \mathcal{C}_t), \tag{8}$$

where p is the number of blocks following the updatable smart contract $S[i]$. The update in the contract is static for a particular period of time, till the next change is initiated. Therefore, for k updates in the smart contract, the cost becomes $Cost_k$:

$$Cost_k = Cost_{PCIHF} + k.Cost_T, \tag{9}$$

where $Cost_{PCIHF}$ is the cost of the first hash with PCIHF. $Cost_T$ is static and does not depend on the number of blocks or transactions. Table IV provides the summarized measurements of complexities calculated for all the frameworks. We measure the complexities in O notation. In the calculation, T is the number of transactions, and n is the number of updates in a smart contract. Comparatively, our proposed LEAGAN is less complex; for each change n, each transaction should execute the change as shown in Figure 3. Thus, our time complexity is $O(T.n)$. Each change n is stored in the status contract; rather than storing the whole contract separately - LEAGAN stores only the changed part in the logic contract. Thus, our space complexity is significantly low and has the value of $O(n)$. From the experimental analysis of 100 block generation with our proposed LEAGAN framework, we infer that PCIHF needs $32ms$. Smart contract modification request verification requires $320ms$, and versioning and RCS combined require $\sim$ $300ms$. Therefore, the overall time requirement for LEAGAN for the successful implementation of a change in the smart contract takes $\sim 652ms$, which is 25% less on average as compared to the existing upgradeable smart contract frameworks. Furthermore, to validate one of the contributions, i.e., reduced storage space, we compare CPU utilization, Input/Output (I/O) disk space, and storage space requirements of our LEAGAN in Table V. From this table, we see that in all the space utilization parameters, our LEAGAN outperforms the existing frameworks due to the strategic use of PCIHF and RCS, which reduces the space requirements significantly. For I/O disk utilization, LEAGAN optimizes data read/write operations by storing only modified contract logic rather than full contract redeployments. The average data size per storage update is $\sim$ 256–512 bytes, depending on the number of logic contract modifications recorded in the status contract. Traditional proxy-based models rewrite entire contract states, consuming $\sim$ 45–50% disk I/O, whereas LEAGAN logs incremental updates ($\sim$ 20–25% disk I/O). Read operations, such as querying contract storage (eth_getStorageAt), typically consume 32–128 bytes per request. Write transactions, involving new contract version storage and incremental hashing, remain within 512–1024 bytes per update cycle, ensuring lower storage costs and efficient blockchain state management.

From the above analysis of LEAGAN, we can comment that LEAGAN is efficient as an upgradeable smart contract framework and exploits the benefits of memory consumption reduction, enhancement of the version management, and security of the smart contract.

D. Solving the research gaps of SOTA

The proposed LEAGAN framework addresses multiple research gaps identified for upgradeable smart contracts, as outlined in Table I. LEAGAN overcomes the centralization risks associated with off-chain trusted deployers by introducing a decentralized verification mechanism - this ensures that updates are validated through a transparent voting system by network peers rather than relying on a single trusted entity.

LEAGAN resolves scalability and proxy selector clash issues observed in Comprehensive-Data-Proxy patterns by employing a status contract that efficiently maps logic contract versions while maintaining immutability in storage contracts, reducing overhead and selector conflicts. Moreover, the bytecode rewriting complexity and compatibility issues of EVMPatch are mitigated in LEAGAN by utilizing incremental hashing (PCIHF) and an RCS; it allows contract modifications without requiring complete redeployment, which improves compatibility and reduces debugging overhead.

The lack of modularity in upgradeability mechanisms like Elysium is tackled by LEAGAN's data-separation strategy, where logic, storage, and status contracts are independently managed. This separation strategy allows for structured and modular updates. In addition, LEAGAN optimizes the on-chain verification overhead of four-tier models by leveraging incremental hashing for contract state tracking to minimize redundant storage and computational load. LEAGAN addresses concerns about access control in proxy mechanisms by using decentralized signature verification and update validation mechanisms. This ensures governance transparency and prevents single-user control over contract updates. High recompilation problems, as in ATOM, are mitigated in LEAGAN by storing and updating only modified contract parts using RCS; this significantly reduces execution time and storage consumption. Differentiated code approaches, which focus on logic without resource optimization, are enhanced in LEAGAN by integrating version control and efficient storage mechanisms, ensuring that updates are applied with minimal blockchain

state bloat. Auto-update frameworks such as LSC, which are limited in generalizability, are improved upon by LEAGAN's flexible status contract architecture; this allows a broad range of decentralized applications to implement and benefit from structured contract versioning. All these solution strategies collectively make LEAGAN a robust and comprehensive solution for smart contract upgradeability, addressing fundamental issues of immutability, version management, security, and computational overhead in blockchain systems.

TABLE IV
COMPLEXITY ANALYSIS

Framework	Time complexity	Space complexity
Proxy [10]	$O(T^n)$	$O(T^n)$
Differentiated code [18]	$O(T^n) + O(n^n)$	$O(T^n)$
Auto-update [19]	$O(T.n)$	$O(T)$
Proposed LEAGAN	$O(T.n)$	$O(n)$

TABLE V
RESOURCE UTILIZATION ANALYSIS

Framework	CPU utilization (%)	I/O disk utilization (%)	Average storage utilization(%)
Proxy [10]	32	54	50
Differentiated code [18]	35	30	45
Auto-update [19]	40	30	38
Proposed LEAGAN	28	25	20

VI. CONCLUSION

In this paper, we introduce LEAGAN as a mutable or upgradeable smart contract framework. LEAGAN is the first smart contract framework to use incremental hash in the data separation method to reduce the memory overhead of smart contract changes. Besides, LEAGAN adds novel features, including status contracts to securely distinguish between logic contracts and storage contracts. Besides, our proposed LEAGAN is innovative in the direction of version management, where LEAGAN stores only the changes in the smart contract and avoids restoring the whole contract multiple times. Security analysis of LEAGAN concludes that LEAGAN and its internal functions are secure. A thorough set of experiments is performed in LEAGAN. Experimental analysis and comparison show that LEAGAN provides 27.3% better throughput, 24.5% better gas consumption, 100% accuracy, and 25% less time-consuming. Thus, LEAGAN is a novel and efficient innovative contract solution for blockchains.

While LEAGAN significantly improves smart contract upgradeability, gas efficiency, and version control, it has certain limitations. LEAGAN may introduce latency in hash computations for frequent updates. RCS efficiently tracks contract versions but may require additional storage management techniques to handle long-term scalability. LEAGAN improves gas consumption; transaction batching and state compression optimization could enhance execution speed and cost efficiency. For future work, we plan to extend LEAGAN capabilities by integrating automated storage pruning, state compression techniques, and parallel execution of contract updates, which can further reduce computation and storage overhead.

ACKNOWLEDGEMENT

This work was supported by the European Commission under the Horizon Europe Programme, as part of the project LAZARUS (https://lazarus-he.eu/) (Grant Agreement no. 101070303).

REFERENCES

[1] A. Vacca, A. D. Sorbo, C. A. Visaggio, G. Canfora, A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges, Journal of Systems and Software, vol. 174, 2021, pp. 1-19, doi: https://doi.org/10.1016/j.jss.2020.110891.

[2] S. Mathur, A. Kalla, G. Gür, M. K. Bohra, M. Liyanage, A Survey on Role of Blockchain for IoT: Applications and Technical Aspects, Computer Networks, vol. 227, 2023, pp. 1-46, doi: https://doi.org/10.1016/j.comnet.2023.109726.

[3] Leng, M. Zhou, J. L. Zhao, Y. Huang and Y. Bian, Blockchain Security: A Survey of Techniques and Research Directions, IEEE Transactions on Services Computing, vol. 15, no. 4, 2022, 2022, pp. 2490-2510, doi: 10.1109/TSC.2020.3038641.

[4] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo and T. Chen, Defining Smart Contract Defects on Ethereum, IEEE Transactions on Software Engineering, vol. 48, no. 1, 2022, pp. 327-345, doi: 10.1109/TSE.2020.2989002.

[5] B. Hu, Z. Zhang, J. Liu, Y. Liu, J. Yin, R. Lu, X. Lin, A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems, Patterns, vol. 2, no. 2, 2021, pp. 1-51, doi: https://doi.org/10.1016/j.patter.2020.100179.

[6] An Introduction to Upgradeable Smart Contracts, available at: https://www.quicknode.com/guides/ethereum-development/smart-contracts/an-introduction-to-upgradeable-smart-contracts/, accessed on: 20-05-2023.

[7] J. Chen, Finding Ethereum Smart Contracts Security Issues by Comparing History Versions, 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1382-1384, 2020, doi: https://doi.org/10.1145/3324884.3418923.

[8] S. Meisami & W. E. Bodell III., A Comprehensive Survey of Upgradeable Smart Contract Patterns, arXiv:2304.03405v1, 2023, pp. 1-7, doi: https://doi.org/10.48550/arXiv.2304.03405.

[9] Ye Liu and Shuo Li and Xiuheng Wu and Yi Li and Zhiyang Chen and David Lo, Demystifying the Characteristics for Smart Contract Upgrades, arXiv: 2406.05712, 2024, pp. 1-12, doi: https://doi.org/10.48550/arXiv.2406.05712.

[10] W. E. Bodell III, S. Meisami, Y. Duan, Proxy Hunting: Understanding and Characterizing Proxy-based Upgradeable Smart Contracts in Blockchains, 32nd USENIX Security Symposium, 2023, pp. 1829 - 1846.

[11] P. Antonino, J. Ferreira, A. Sampaio, A. W. Roscoe, Specification is Law: Safe Creation and Upgrade of Ethereum Smart Contracts, Software Engineering and Formal Methods, Lecture Notes in Computer Science, vol. 13550, Springer, Cham, 2022, pp. 227-243, doi: https://doi.org/10.1007/978-3-031-17108-6_14.

[12] V. C. Bui, S. Wen, J. Yu, X. Xia, M. S. Haghighi, Y. Xiang, Evaluating Upgradable Smart Contract, IEEE International Conference on Blockchain (Blockchain), 2021, pp. 252-256, doi: 10.1109/Blockchain53845.2021.00041.

[13] M. Rodler, W. Li, and G. O. Karame, L. Davi, EVMPatch: Timely and automated patching of ethereum smart contracts, 30th USENIX Security Symposium, 2021, pp. 1289-1306.

[14] C. F. Torres, H. Jonker, R. State, Elysium: Context-Aware Bytecode-Level Patching to Automatically Heal Vulnerable Smart Contracts, RAID '22: Proceedings of the 25th International Symposium on Research in Attacks, Intrusions, and Defenses, 2022, pp. 115-128, doi: https://doi.org/10.1145/3545948.3545975.

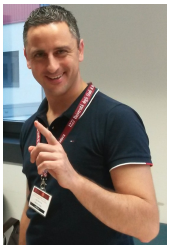
[15] Z. Du, H. Cheng, Y. Fu, M. Huang, L. Liu, Y. Ma, A Four-Tier Smart Contract Model with On-Chain Upgrade, Security and Communication Networks, vol. 2023, 2023, pp. 1-12, doi: https://doi.org/10.1155/2023/8455894.

[16] M. Salehi, J. Clark, M. Mannan, Not so Immutable: Upgradeability of Smart Contracts on Ethereum. Financial Cryptography and Data Security, FC 2022 International Workshops, Lecture Notes in Computer Science, vol. 13412, Springer, Cham, 2023, pp. 539-554, doi: https://doi.org/10.1007/978-3-031-32415-4_33.
[17] T. Li, Y. Fang, Z. Jian, X. Xie, Y. Lu and G. Wang, ATOM: Architectural Support and Optimization Mechanism for Smart Contract Fast Update and Execution in Blockchain-Based IoT, IEEE Internet of Things Journal, vol. 9, no. 11, 2022, pp. 7959-7971, doi: 10.1109/JIOT.2021.3106942.
[18] Y. Huang, Q. Kong, N. Jia, X. Chen and Z. Zheng, Recommending Differentiated Code to Support Smart Contract Update, IEEE/ACM 27th International Conference on Program Comprehension (ICPC), Montreal, Canada, 2019, pp. 260-270, doi: 10.1109/ICPC.2019.00045.
[19] W. Shao, Z. Wang, X. Wang, K. Qiu, C. Jia, C. Jiang, LSC: Online auto-update smart contracts for fortifying blockchain-based log systems, Information Sciences, vol. 512, 2020, pp. 506-517, doi: https://doi.org/10.1016/j.ins.2019.09.073.
[20] M. Bellare, O. Goldreich, S. Goldwasser, Incremental Cryptography: The Case of Hashing and Signing, Advances in Cryptology – Crypto 94 Proceedings, Lecture Notes in Computer Science Vol. 839, Springer-Verlag, Y. Desmedt, ed., 1994, pp. 216-233.
[21] B. M. Goi, M. U. Siddiqi, H.T. Chuah, Computational complexity and implementation aspects of the incremental hash function, IEEE Transactions on Consumer Electronics, vol. 49, no. 4, 2003, pp. 1249-1255, doi: 10.1109/TCE.2003.1261226.
[22] National Institute of Standards and Technology, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, FIPS PUB 202, 2015, available at: https://csrc.nist.gov/publications/detail/fips/202/final, accessed on: 10–3-2023.
[23] Recursive Length Prefix (RLP) Encoding, available at: https://ethereum.github.io/execution-specs/autoapi/ethereum/rlp/index.html#introduction, accessed on: 05-05-2023.
[24] Don Bolinger, Tan Bronson, Applying RCS and SCCS - From Source Control to Project Control. O'Reilly, 1995.

AUTHOR BIOGRAPHY

Gulshan Kumar (Senior Member, IEEE) has received his Ph.D. degree in Computer Science and Engineering from Lovely Professional University, Punjab, India in 2017. He is currently working as a Postdoc at the University of Padua, Italy and an Associate Professor at Lovely Professional University, Punjab, India. His current research interests include cyber-physical systems, blockchain, edge, and cloud computing, wireless sensor networks, and optimization techniques.

Rahul Saha (Member, IEEE) has received a Ph.D. degree in cryptography from Lovely Professional University, Punjab, India. He is currently working as an Associate Professor with Lovely Professional University and also as a Postdoc at the University of Padua, Italy. His research interest includes network security, cryptography, blockchain, and DLTs, IoT security.



Mauro Conti (Fellow IEEE) is Full Professor at the University of Padua, Italy. His main research interest is in the area of Security and Privacy. He is a Fellow IEEE and Senior Member, ACM. He is a member of the Blockchain Expert Panel of the Italian Government. He is a Fellow of the Young Academy of Europe.

William Johnston Buchanan OBE FRSE CEng PFHEA is a Scottish computer scientist. He currently leads the Blockpass ID Lab and the Centre for Cybersecurity, IoT and Cyberphysical at Edinburgh Napier University. He is a professor in the School of Computing, Engineering and the Built Environment. Bill was awarded an OBE in 2017 for services to cybersecurity, and elected as a Fellow of the Royal society of Edinburgh (FRSE) in 2024.