








SmartUpdater: Enabling Transparent, Automated, and Secure Maintenance of Stateful Smart Contracts

Xiaoli Zhang , Yiqiao Song , Yuefeng Du , Chengjun Cai , *Member, IEEE*, Hongbing Cheng ,
Ke Xu , *Fellow, IEEE*, and Qi Li , *Senior Member, IEEE*

I. INTRODUCTION

Abstract—Smart contracts in the Ethereum system are stored tamper-resistant, complicating necessary maintenance for offering new functionalities or fixing security vulnerabilities. Previous contract maintenance approaches mainly focus on logic modification using delegatecall-based patterns. While popular, they fail to handle data state updates (like storage layout changes), leading to impracticality and security risks in real-world applications. To address these challenges, this paper introduces SmartUpdater, a novel toolchain designed for transparent, automated, and secure maintenance of stateful smart contracts. SmartUpdater employs a hyperproxy-based contract maintenance pattern, where the hyperproxy serves as a constant entry and ensures that any state/logic modifications remain transparent to end users. SmartUpdater automates the maintenance process in terms of development streamlining, gas cost efficiency, and state migration verifiability. In extensive evaluations, we show that SmartUpdater can reduce gas consumption in contract maintenance compared with actual maintenance approaches. The evaluations point out the potential of SmartUpdater to significantly simplify the maintenance process for developers.

Index Terms—Stateful smart contract, upgradable contract, state migration, EVM.

Received 24 June 2024; revised 7 February 2025; accepted 1 March 2025. Date of publication 6 March 2025; date of current version 18 April 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62302452, Grant 62132011, Grant 62425201, Grant 62072407, and Grant 62202398; in part by Zhejiang Provincial Natural Science Foundation of China under Grant LQ23F020019 and Grant LZ24F020007; in part by the “Leading Goose Project Plan” of Zhejiang Province under Grant 2022C01086 and Grant 2022C03139. Recommended for acceptance by M. Kechagia. (*Corresponding authors: Xiaoli Zhang; Hongbing Cheng.*)

Xiaoli Zhang is with the Department of Computer Science, Zhejiang University of Technology, Hangzhou 310023, China, and also with the School of Computer & Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China (e-mail: xiaoli.z@outlook.com).

Yiqiao Song and Hongbing Cheng are with the Department of Computer Science, Zhejiang University of Technology, Hangzhou 310023, China (e-mail: 201806062522@zjut.edu.cn; chenghb@zjut.edu.cn).

Yuefeng Du is with the Computer Science Department, City University of Hong Kong, Hong Kong, SAR 999077, China (e-mail: yf.du@cityu.edu.hk).

Chengjun Cai is with the Computer Science and Information Technology Center, City University of Hong Kong (Dongguan), Dongguan 523808, China (e-mail: chengjun.cai@cityu.edu.cn).

Ke Xu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: xuke@tsinghua.edu.cn).

Qi Li is with the Institute of Network Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: qli01@tsinghua.edu.cn).

Digital Object Identifier 10.1109/TSE.2025.3548730

THE popularity of decentralized applications (DApps), particularly in decentralized finances (DeFi), has surged, with the DApps market valued at \$25.63 billion in 2022 and projected to reach \$70.82 billion by 2030 [43]. These DApps are fundamentally built on smart contracts, known for their immutability – a feature that is critical for execution security in blockchain systems. However, similar to traditional software, smart contracts require maintenance after deployment to address security vulnerabilities and functional requirements (as a real-world example shown in Fig. 1). As indicated by a study [54], as of 2022, 46% of smart contracts deployed on Ethereum necessitate maintenance, reflecting a prevalent practice.

In response to these problems, one research line aims to develop flawless smart contracts [52], [59], yet is quite challenging due to the variety of defect types and the limited testing capabilities for covering all possible scenarios [31], [60]. Instead, the other research field focuses on smart contract maintenance [14], [37], [44]. As shown in Fig. 2(a), one intuitive solution is renewing entire contracts [36], but this approach requires end users to use the new contract address to send transactions. If the old contract’s address is mistakenly used, it may trap funds and lead to financial losses [12]. To avoid affecting user experiences, the call-based contract maintenance pattern has been proposed [34] (see Fig. 2(b)). It decouples execution logic (i.e., the code that implements the contract’s functionality) and state variables (i.e., the data stored in the contract’s storage) into separate contracts, and utilizes logic contracts as a constant user entry point. This way, any modifications to *state contracts* are transparent to end users. At the same time, a delegatecall-based pattern [14] is proposed (see Fig. 2(c)). It separates state and logic contracts and only allows for user-transparent modifications to *logic contracts* by using state contracts as the constant user entry point.

Prior contract maintenance approaches are not sufficient in practice due to the following reasons. First, both state variables and the logic of contracts would change unpredictably [17], yet prior approaches cannot always keep those changes transparent to end users, i.e., the contract address that end users assess would change. This problem possibly introduces black-hole risks of deprecated contracts and leads to financial losses [12]. Second, there are sophisticated state update requirements that would change storage layout, e.g., adding new

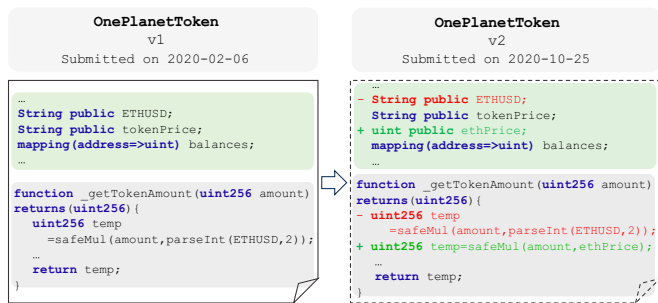


Fig. 1. A real-world example of contract maintenance. The OnePlanetToken contract updates its price management functionality in the new version (0x73Ea...DA3b).

variables or removing or changing the types of the current state variables, as shown in Fig. 1. These requirements account for 60% of all maintenance requirements with state update operations [50]. Unfortunately, existing approaches fail to guarantee the correctness of state variables in these cases. For example, as shown in Fig. 2(b) and 2(c), the call-based method faces the problem of recovering values of unchanged variables while the delegatecall-based solution would incur storage collision between new and old state variables and lead to huge financial losses (e.g., a \$1.6 million loss is caused due to the incorrect maintenance of Pike Finance contracts [13]). Therefore, a pivotal research question arises: “How can we devise a stateful contract maintenance tool that can support both logic and state modifications while keeping the maintenance process transparent to end users?”

The SmartUpdater Design. In the paper, we introduce SmartUpdater, a novel toolchain designed for user-transparent maintenance of stateful smart contracts. SmartUpdater adopts an innovative hyperproxy-based contract maintenance pattern, which decouples logic and state contracts while leveraging a “hyperproxy” contract managing logic/state updates and serving as the consistent entry for end users. Once user calls arrive, as shown in Fig. 2(d), the hyperproxy dynamically routes the calls to the appropriate underlying contracts. Therefore, any changes to the logic or states would not affect the access addresses of end users, preserving a seamless user experience and contract maintenance security.

Developing and deploying smart contracts that adhere to the hyperproxy-based contract maintenance pattern faces several practical challenges throughout the contract maintenance process. First, adapting smart contracts with complex execution logic and state variables to meet diverse maintenance requirements is often labor-intensive and prone to errors. To address this, SmartUpdater introduces an intuitive development interface featuring a Solidity-like domain-specific language (DSL), allowing developers to streamline contract development and maintenance. Second, in resource-expensive blockchain environments, contract maintenance typically introduces expensive gas costs due to massive contract deployment and state migration. To ensure minimal gas expenditure, we propose an optimization-oriented contract generation module that strategically divides a DSL-written contract into sub-contracts and

tries to make only the necessary components be modified in each maintenance. Third, correctly recovering states in updated contracts is necessary yet difficult as varied-type states are stored in different manners and hard to collect. Regarding this, we propose a publicly verifiable state migration method that leverages the unique characteristics of different state types and employs a SNARK-based approach to guarantee the correct migration of sophisticated states.

We conducted extensive experiments using 1,355 real-world contract maintenance cases. The experimental results show that developers complete contract development using SmartUpdater in under 10 minutes on average. Throughout the whole contract maintenance process, 72% contracts successfully reduce gas consumption compared to the usage of actual contract maintenance approaches, saving 711K gas (\$78.5) on average. Additionally, SmartUpdater’s refined SNARK-based verifiable state migration design decreases on-chain verification costs from over 4300M gas to 1024K gas.

Contributions. In summary, the contributions are as follows:

- We introduce SmartUpdater, a novel toolchain for user-transparent maintenance of stateful smart contracts. It particularly devises a hyperproxy-based contract maintenance pattern which leverages a hyperproxy contract as a constant entry for end users while managing the updates of the logic and state contracts.
- Considering the laborious, costly, and error-prone contract development and deployment pipelines, SmartUpdater includes an expressive programming module, an optimization-oriented contract generation module, and a publicly verifiable state migration module. Those modules work together to generate or maintain contracts following the hyperproxy-based contract maintenance pattern, particularly with the features of development-streamlining, gas-cost-efficiency, and state-migration-verifiability.
- We carried out extensive experiments using 1,355 real-world contract maintenance cases, demonstrating the effectiveness and efficiency of SmartUpdater. We also released the source code of SmartUpdater at Github [1].

II. BACKGROUND

This section provides a brief introduction to the necessary background knowledge, organized into three parts:

Blockchain. We focus on the Ethereum protocol, which is the first blockchain protocol supporting smart contracts. At its core, the Ethereum blockchain operates as a decentralized ledger, made up of a series of blocks. As shown in Fig. 3, each block, symbolized as blk , carries a header, blk_{Head} , housing vital metadata like a hash pointer to the prior block and a summary of transaction results through the Merkle Patricia tree root. Outcomes of these transactions, termed receipts, capture events during a smart contract’s execution and are chronologically stored in the *Logs* array. Each receipt within *Logs* contains the topic identifier, which facilitates efficient event retrieval. To quickly confirm the occurrence of particular events, a *LogsBloom* filter is embedded into both the block header and the receipt.

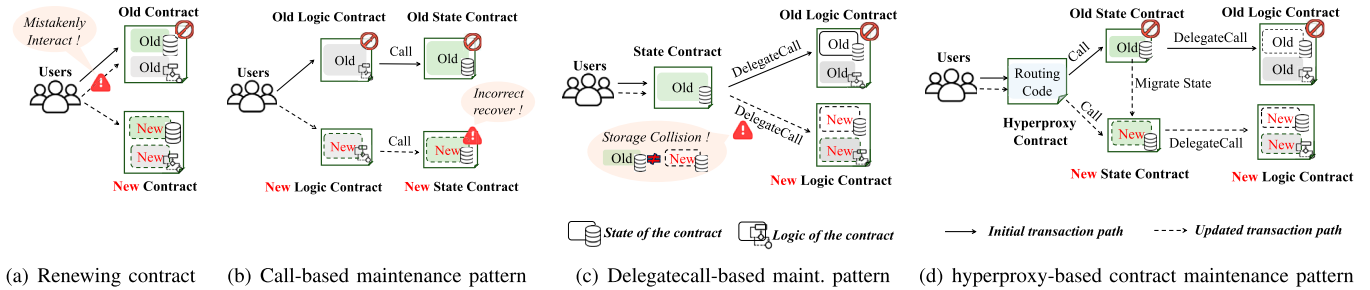


Fig. 2. Comparison of existing smart contract maintenance approaches.

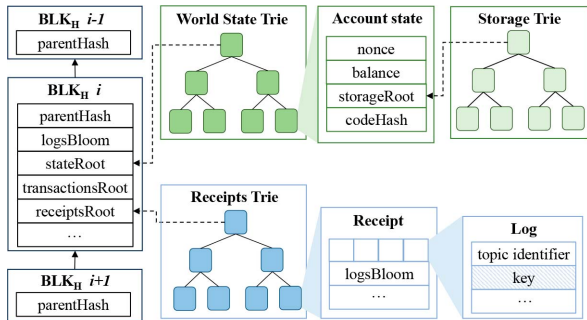


Fig. 3. Blockchain storage structure.

EVM Storage. Ethereum Virtual Machine (EVM) is engineered for executing smart contracts. It boasts short-term memory for temporary data and persistent storage designed for global states. This storage is conceived as a vast array, with a capacity of up to 2^{256} slots. Each slot comprises 32 bytes. According to the order of state declaration within a contract, data is stored beginning from the inaugural slot [8]. Storage of state variables follows two distinctive patterns:

Deterministic storage layout: Deterministic storage is straightforward in its approach: state variables are allocated to fixed storage slots based on the order in which these state variables are declared within the contract. EVM employs this type of storage layout for basic data types such as integer, boolean, and fixed-size arrays.

Dynamic storage layout: In contrast, the dynamic storage layout manages state types whose size or structure might change over time, like dynamic array and mapping. For these types, a primary slot called the *baseslot* is initially designated according to state declaration orders. Then, the actual storage location of each data is computed based on *baseslot* and state types. For a dynamic array, *baseslot* retains the array's length and the data elements are continuously stored starting from the slot indexed by $\text{KEC}(\text{baseslot})$. For a mapping-type state, each data is stored at the slot $\text{KEC}(h(k) \cdot \text{baseslot})$ where \cdot represents concatenation and h is a type-dependent function applied to the key k . Note that the keys are usually not stored in the storage, rendering them inaccessible from the blockchain storage.

Verifiable Computation. Public verifiable computation (VC) schemes verify complex computations without fully re-executing them [39]. Within a set security framework, two

keys are generated for any computation F . One executes F and the other verifies the result's authenticity. When a computation is performed, the outcome is paired with a cryptographic proof of its validity, enabling clients to confirm correctness without revisiting every step.

III. RELATED WORK

This section outlines some work related to smart contract maintenance, contract vulnerability detection, and verifiable computation in the blockchain.

Smart Contract Maintenance. Several studies have tackled the challenge of smart contract maintenance [11], [29], [32], [44]. Among them, EVMPatch [44] and Aroc [29] primarily focus on the logic update in the smart contract and ignore how to deal with data including assets and states. SolSaviour [32] uses a TEE cluster for secure asset migration, but overlooks states with complex types. Smartmuv [11], an automatic static analysis tool, extracts states from the storage and approximates the origin of the keys used in the mapping. However, its precision for mapping state extraction is only 95.7%. In contrast, this paper presents an automated tool designed to adeptly handle both state and logic updates, guaranteeing precise migration of intricate state types.

Contract Vulnerability Detection. Existing contract vulnerability detection tools fall into three categories: symbolic execution tools [25], [48], static analysis tools [45], [49], and dynamic analysis tools [18], [24], [57]. For instance, DEFINERY [48] uses symbolic execution to analyze vulnerabilities in smart contracts. Securify [49] is a static analysis tool that can prove whether the contract behavior is safe with respect to a given property. EVM-Shield [57] adopts a fine-grained access control and monitors behaviors of transactions to protect smart contracts in real-time. The above studies are orthogonal to our focus.

Verifiable Computation in Blockchain. Numerous studies have harnessed verifiable computation, such as zero-knowledge proofs, to bolster security. These investigations predominantly target dilemmas surrounding transaction trustworthiness [15], [33], user privacy safeguards [46], [47], and interactivity issues [55]. For instance, Shunli et al. [33] deploy a zero-knowledge scheme to authenticate the legitimacy of transactions. Conversely, Zapper [47] underscores the preservation of user anonymity, shielding both user identities and their corresponding engagements. In a related vein, zkBridge [55] taps into

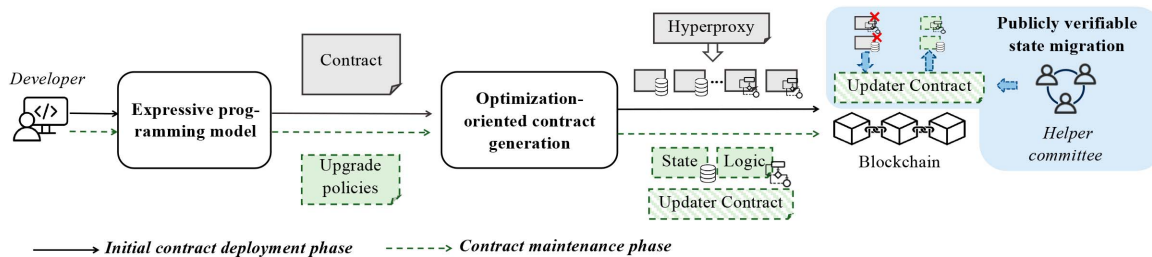


Fig. 4. The workflow of SmartUpdater.

zero-knowledge proofs to erect a secure cross-chain bridge. Interestingly, the realm of verifiable computation has remained untouched in the context of contract maintenance. Our study aims to bridge this gap by leveraging such techniques to establish a tamper-resistant contract maintenance framework.

IV. DESIGN AND IMPLEMENTATION

This section introduces design goals, design overview, design details of each component, and corresponding implementations.

A. Design Goals

User-transparent Contract Maintenance. The primary goal is to keep the contract maintenance process invisible to the end users, i.e., ensuring that end users’ access addresses remain constant, regardless of any changes in the contract’s state variables or execution logic. This feature not only ensures seamless user experience by preserving an unchanged contract address and prevents accidental access to the deprecated contracts, but also enhances contract composability with others¹, improving DApps’ compatibility and flexibility.

Streamlined Contract Development. When facing sophisticated state/logic updating requirements for old contracts with various state variables, streamlining the contract maintenance process is essential for boosting the adoption of a contract maintenance toolchain and practices among developers [42]. More specifically, an intuitive development interface that can effectively integrate with popular contract language (e.g., Solidity) and adapt to developers’ diverse maintenance strategies is highly desired.

Optimal Maintenance Cost. Contract maintenance typically introduces expensive gas costs for deploying new contracts and migrating states in the resource-expensive blockchain environment. For example, state migration for only 5,000 user accounts in a smart contract would cost more than 60M gas costs (\$6235) [36]. The costs would grow significantly if frequent maintenance is needed. Therefore, minimizing contract maintenance gas costs—especially those related to contract deployment and state variable migration—is necessary.

Correctness-guaranteed State Migration. Once new state contracts are deployed, it is essential to ensure correct migration of state variables from old contracts to new ones. This property

¹Note that a contract can be externally accessed by end users and other contracts. For simplicity, we do not distinguish them and use the term “end user” throughout the paper.

must support the migration of all types of state variables, particularly those that are difficult to retrieve (e.g., mapping-type state), ensuring state correctness and consistency throughout the process.

B. Design Overview

We present a smart contract maintenance tool called SmartUpdater that can support both state and logic updates while maintaining uninterrupted access to the smart contract for end users. Specifically, SmartUpdater introduces a novel hyperproxy-based contract maintenance pattern (see Sec. IV-C). As shown in Fig. 2(d), this pattern leverages a hyper-proxy contract that is the constant entry for end users and manages the underlying business contracts, such that any logic and state updates are invisible to end users.

To implement the pattern while meeting the other design goals, we describe the workflow according to two typical phases in contract maintenance:

Initial Contract Deployment Phase: As depicted in Fig. 4, SmartUpdater provides an expressive programming model for contract developers to write smart contracts (see Sec. IV-D). The domain-specific language (DSL) particularly integrates with the popular contract language, Solidity, and supports easy specification of logic/state update policies for streamlining contract development and maintenance. Then, the DSL-written contracts are processed by the optimization-oriented contract generation module (see Sec. IV-E). By balancing the gas costs of contract deployment and those of contract maintenance, the contracts are automatically modularized into a set of sub-contracts. Finally, this module also generates a specific hyper-proxy contract to coordinate all these sub-contracts, which are further deployed on the blockchain system.

Contract Maintenance Phase: During contract maintenance, developers specify logic or state modification policies using the DSL. These policies are processed by the optimization-oriented contract generation module, which generates the corresponding new sub-contracts required for maintenance. Once these new sub-contracts are deployed, a publicly verifiable state migration module is activated to ensure the correct migration of any types of state variables from the old to the new version in a decentralized manner (see Sec. IV-F). It involves a helper committee to facilitate restoring sophisticated states with dynamic storage layout, e.g., the mapping type.

In a nutshell, SmartUpdater is a toolchain that includes three modules: an expressive programming module,

an optimization-oriented contract generation module, and a publicly verifiable state migration module. To support user-transparent logic/state updates, SmartUpdater generates contracts following the hyperproxy-based contract maintenance pattern.

Trust Assumptions. We outline our trust assumptions for key components of SmartUpdater. We trust the blockchain system for its integrity and availability [32]. Smart contracts deployed on the blockchain system will operate as designed without any risk of external interference. Meanwhile, our model assumes that the contract developers are responsible for both the deployment and any future maintenance of smart contracts. For simplicity, we assume these developers are trustworthy, since malicious developers, that might maliciously alter key parameters for personal benefits such as stealing token increases, can be defeated by existing decentralized contract governance methods [24], [32]. Last, SmartUpdater’s reliability hinges on the presence of at least one honest node in the helper committee. The honesty of the node is crucial for the accurate and uninterrupted operation of the verifiable state migration process.

C. Hyperproxy-Based Contract Maintenance Pattern

To keep the contract maintenance process transparent to end users, SmartUpdater employs an independent contract, i.e., hyperproxy contract, as the constant entry, which further forwards user requests to the backward business contracts. Therefore, any logic/state updates are invisible to external users or contracts. At the same time, to simplify the management of states and logic of the business contracts, we follow the widely adopted delegatecall-based patterns [14], [44] to partition the logic and states of the business contracts and make the state contracts delegate calls to the corresponding logic contracts. Finally, when user transactions arrive, the hyperproxy contract redirects the transaction to the appropriate state contracts, which delegate the transactions to the logic contract for function execution.

In more detail, the hyperproxy consists of three main components: 1) a state-logic mapping that records the relationships between the state contract addresses and external functions’ identifiers² that can be accessed by transactions; 2) a contract-setting function that is responsible for setting newly patched contracts’ addresses in the above mapping; 3) a *fallback*-based transaction-forwarding function that searches the matched underlying contract from the above mapping based on the user transactions and automatically redirects outside user calls to the corresponding state contract. To prevent malicious access to the state contract by bypassing the hyperproxy contract, the transaction-forwarding function uses the *call* method to send transactions to the state contracts. This ensures that state contracts can directly verify whether the transactions they receive originate from the hyperproxy.

Fig. 5 illustrates a hyperlayer, a new state contract that replaces an old one, and a logic contract. When a user issues a

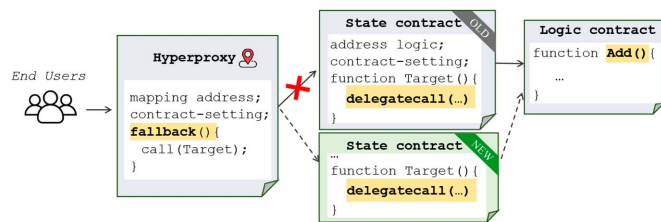


Fig. 5. An example of how the hyperproxy contract manages updates of state and logic contracts while keeping the changes transparent to outside parties.

transaction to invoke a call function called *Add* by providing its function selector (e.g., *0x67e06858*) along with the hyperproxy’s address, the *fallback* function of the hyperproxy is automatically executed. Thereafter, the hyperproxy searches the mapping and forwards the user’s call to a specific state contract that can access the function of *Add* in the logic contract. The state contract in turn initiates a *delegatecall* to the target *Add* function in the logic contract. During a state update, a new state contract replaces the old one. Simultaneously, the contract-setting function is triggered, updating the state-logic mapping with the new state contract’s address in place of the old one. In this way, despite state or logic contracts being replaced, users can always interact with the constant address of the hyperproxy contract.

Regarding who can perform contract maintenance, hyperproxy can use an admin address (e.g., the developer address) or employ existing smart contract based voting schemes [24], [32] for secure decentralized governance. Specifically, for each smart contract, a committee consisting of eligible voters is responsible for voting to either accept or reject a maintenance requirement for the contract. If it is approved by the majority of voters, the new contract address will be set for corresponding function selectors in the mapping.

D. Expressive Programming Model

SmartUpdater’s programming model aims to streamline the contract development and state/logic modification policy definition for contract developers, especially in line with subsequent optimization of maintenance costs. Although Solidity is widely used for writing smart contracts, we still encounter two challenges in maintaining stateful contracts: 1) State storage characteristics, such as storage size, directly impact future update costs, which are essential for optimal contract deployment but can only be determined at runtime. 2) Different types of state variables require varying maintenance actions, from simple renaming to complete data layout reconstruction, making it difficult to define precise maintenance policies.

In response, SmartUpdater resorts to the domain-specific knowledge of contract developers and enables them to estimate and annotate storage features of various states before deployment. Meanwhile, inspired by database operations [51], SmartUpdater provides database-like update interfaces to support modifications of states of different types.

²Note that a function identifier in the context of smart contracts is also called a function selector, which is the first four bytes of the hash of the function signature.

```

1 contract CreditDAO {
2   struct Election{
3     // @mag 100 @bat 2
4     mapping(address => bool) candidates;
5     mapping(address => bool) userHasVoted;
6     address maxVotes;
7     uint numMaxVotes;
8   }
9   // @mag 100
10  mapping(uint => Election) public elections;
11 }

```

(a) Original CreditDAO contract [19] written with SmartUpdater's DSL

```

1 @StatePolicy 0xdAC1...1ec7{
2   CREATE Participant{
3     INSERT (isCandidate, bool, -, -);
4     INSERT (hasVoted, bool, -, -);
5   };
6   ALTER Election{
7     INSERT
8     (userMap, mapping(address=>Participant),
9     -, -);
10    UPDATE (candidates, -, -, -) AS
11    (userMap.isCandidate, -, -, -);
12    UPDATE (userHasVoted, -, -, -) AS
13    (userMap.hasVoted, -, -, -);
14  };
15 }

```

(b) State modification policy for CreditDAO maintenance

```

1 contract CreditDAO {
2   struct Election{
3     mapping(address => Participant) userMap;
4     address maxVotes;
5     uint numMaxVotes;
6   }
7   struct Participant{
8     bool isCandidate;
9     bool hasVoted;
10  }
11  mapping(uint => Election) public elections;
12 }

```

(c) New CreditDAO contract generated from the state modification policy

Fig. 6. SmartUpdater's capability in automatically generating modified contract using the DSL-written contract and policy.

Contract Development With State Annotation. The DSL provides an intuitive development interface, enabling developers to write smart contracts as if they were written in Solidity. Particularly, it includes a grammar that allows developers to declare state storage attributes, such as the approximate storage size of one state. These attributes can be estimated by analyzing contracts with similar usage patterns and are utilized by the optimization-oriented contract generation module to determine how states should be divided into different sub-contracts, minimizing contract deployment and maintenance gas costs. The storage attributes for states with deterministic storage layout (e.g., fixed-size array-type) can be easily obtained via static analysis (details presented in Sec. IV-E). However, the storage attributes for states with dynamic storage layout (e.g., dynamic

$$\begin{aligned}
 \text{Policy} &::= \text{@LogicPolicy } \text{Address } \{ \text{Code} \} \\
 &\quad | \text{@StatePolicy } \text{Address } \{ \text{Stmt}^+ \} \\
 \text{Stmt } S &::= \text{InsStmt} \mid \text{DelStmt} \mid \text{UpdStmt} \mid \text{CreStmt} \\
 &\quad | \text{AltStmt} \mid S; S \\
 \text{AltOp } A &::= \text{InsStmt} \mid \text{DelStmt} \mid \text{UpdStmt} \mid A; A \\
 \text{InsStmt} &::= \text{INSERT } (i, t, v, m) \\
 \text{DelStmt} &::= \text{DELETE } (i, t, v, m) \\
 \text{UpdStmt} &::= \text{UPDATE } (i, t, v, m) \text{ AS } (i, t, v, m) \\
 \text{CreStmt} &::= \text{CREATE } \{ \text{InsStmt}^+ \} \\
 \text{AltStmt} &::= \text{ALTER } \{ A^+ \} \\
 \text{Address} &::= \text{0xHexString} \\
 \text{HexString} &::= [0-9a-fA-F]^+ \\
 i \in \text{String} \quad t \in \text{Type} \quad v \in \text{Value} \quad m \in \text{Modifier}
 \end{aligned}$$

Fig. 7. Syntax of the logic/state modification policy in SmartUpdater's expressive programming model. + denotes the previous construction appears once or multiple times.

array-type and mapping-type) are unpredictable and can only be exactly determined at runtime, which is our estimation and annotation focus.

SmartUpdater offers two inline annotation methods to specify storage attributes for states with dynamic storage layout: single-state annotation and batch-state annotation. In the single-state annotation, a developer can specify the state size using a unique identifier @mag, as illustrated in line 10 of the example program (Fig. 6(a)). Meanwhile, when multiple states have a similar size, the batch-state annotation is employed via another identifier @bat. For example, line 3 signifies that the subsequent two states have an expected size of 100 slots in the storage. In addition, to facilitate more efficient maintenance, developers can specify the state update probability via an identifier @prob, such that variables with high update probability might be placed separately to avoid frequent migration of unchanged states. We notice that these annotations are only estimates and are used for the estimation of contract maintenance gas consumption.

Policy Definition With Database-like Update Interfaces.

In SmartUpdater, developers can use our DSL to specify logic/state modification policies of smart contracts, easily defining contract maintenance requirements. Following the well-known EBNF grammar [40], the syntax of the policy is defined in Fig. 7. To distinguish policy types—those for modifying logic or state—the DSL provides two identifiers: @LogicPolicy for logic updates and @StatePolicy for state updates. Each policy is uniquely associated with a specific smart contract address, ensuring the precise identification of the contract that requires maintenance. In the logic modification policy, SmartUpdater accommodates the direct submission of the revised code to update smart contract logic under @LogicPolicy, similar to [44]. Under @StatePolicy identifier, the state modification policy allows developers to easily describe the state update operations for diverse state types. This

policy is composed of multiple operation statements denoted as *Stmt*, each comprising a modification type indicated by an action keyword and a specified target state. The target state is represented via four tuples, formally as (*identifier*, *type*, *value*, *modifier*), and “-” means dummy. In the following, we illustrate the five modification types in the DSL through a practical example involving complex state modifications:

a) INSERT is used to insert new states into new contracts. As an example shown in lines 3-4 of Fig. 6(b), two bool-type states called *isCandidate* and *hasVoted* are inserted.

b) DELETE is specified when removing old states specified with state identifiers. For example, “DELETE (*i_{old}*, -, -, -)” describes that *i_{old}* is deleted in the new contract.

c) UPDATE and AS are used to update old states’ attributes as new states. Examples are shown in lines 8-9 of Fig. 6(b).

d) CREATE is to create a new struct-type state whose element variables are set via INSERT. Lines 2-5 in Fig. 6(b) show an example of creating a struct called *Participant*.

e) ALTER is used for refactoring structure-type states, which would include the first three types of operations to specify how internal elements are modified. As an example shown in lines 6-10, we refactor the *Election* struct by inserting a new element called *userMap* with the type of mapping, changing element variables of *candidate* and *userHasVoted* to the *isCandidate* and *hasVoted* fields in *userMap*.

E. Optimization-Oriented Contract Generation

In a resource-expensive blockchain system like the Ethereum system, we face the following dilemma when designing optimal contract maintenance strategies: putting all states into one contract introduces small gas costs for contract deployment (a base deployment cost of 53000 gas), yet requires heavy maintenance costs due to the migration of all states even if only one state needs to be updated. In contrast, putting a state into one single contract would not incur extra state migration costs yet introduce heavy gas costs for multiple contract deployment. Worse, states are interdependent and cannot be arbitrarily divided into different contracts. Therefore, balancing the contract deployment and maintenance costs while managing massive states is a key challenge.

To tackle the challenge, we devise an optimization function that determines which states are allocated into which state sub-contracts to minimize the costs for overall contract deployment and maintenance.

Optimal Contract Modularization. Let $X_{N_s \times N_c}$ denote an *allocation matrix* to record the relationships between states and sub-contracts where N_s is the number of states in the raw contract and N_c is the *upper bound* of the number of state sub-contracts. Each element, e.g., X_{ij} , is 0 or 1 which represents the association of the i -th state s^i with the j -th state sub-contract c^j . In the optimization, we set each value of $X_{N_s \times N_c}$ to be random numbers within $[0, 1]$ and try to obtain the deterministic modularization results, i.e., 0/1 for state modularization. The objective function is defined as:

$$\min_X D_{ini} + \sum_{i=1}^{N_s} p_i \times (D_{upg}^i + M_{upg}^i), \quad (1)$$

TABLE I
SYMBOLS USED IN OPTIMAL CONTRACT MODULARIZATION

s^i	i -th state
c^j	j -th state sub-contract
p_i	Probability of s^i update
D_{mi}	Deployment cost of the raw contract
D_{upg}^i	Deployment cost required for updating s^i
M_{upg}^i	Total migration cost required for updating s^i
d_i	Contract storage cost of s^i
a_i	Base cost and size-related cost of c^j
m_i	Migration cost of s^i

subject to:

$$A \odot (X - X^T) = T, A_{ij} \in [0, 1] \quad (2)$$

$$XC_1 = C_2 \quad (3)$$

where:

$$D_{mi} = \text{IniDepCost}(\mathbf{a}, \mathbf{d}, X) \quad (4)$$

$$D_{upg}^i = \text{UpgDepCost}(\mathbf{a}, \mathbf{d}, \mathbf{s}^i, X), i \in [1, N_s] \quad (5)$$

$$M_{upg}^i = \text{MigrateCost}(\mathbf{m}, \mathbf{s}^i, X), i \in [1, N_s] \quad (6)$$

The optimization objective, as previously described, is to minimize the overall costs of contract deployment and maintenance. The former refers to the initial deployment costs while the latter encompasses the costs of deploying newly amended contracts and migrating states from the initial versions. Thus, the objective function includes three parts which are weighted by the hyperparameters (i.e., $\mathbf{p} = [p_1, \dots, p_{N_s}]$) to accommodate different state update requirements, as shown in Equation (1). Note that \mathbf{p} represents the state update probability and each value in it is defaulted to 1 if not specified by the developers in the DSL-written contract.

Initial contract deployment cost term D_{mi} : We estimate deployment costs of all state sub-contracts before any maintenance in Equation (4). Note that the primary components of a contract’s deployment cost comprise three aspects: the base cost, the contract size-related cost, and the storage cost [53]. To present more clearly, we denote the base cost and contract size-related cost collectively by $\mathbf{a} = [a_1, \dots, a_{N_c}]$ and the contract storage cost of each state is denoted as $\mathbf{d} = [d_1, \dots, d_{N_s}]$. Thus, the detail of initial contract deployment cost is as follows:

$$\text{IniDepCost}(\mathbf{a}, \mathbf{d}, X) = \sum_{j=1}^{N_c} \left(a_j + \sum_{i=1}^{N_s} \sum_{t=1}^{N_c} X_{it} \times d_t \right) \quad (7)$$

New contract deployment cost term D_{upg}^i : In Equation (5), we explore the deployment costs incurred by updating the states (e.g., s^i). Similar to the above term, for one sub-contract containing a state waiting to be updated (e.g., s^i), there include a base cost plus the contract size-related cost (e.g., $X_{ij} \times a_j$) and the storage costs (e.g., $X_{ij} \times X_{tj} \times d_t$). Therefore, for all updated states, the deployment costs can be formulated as follows:

$$\text{UpgDepCost}(\mathbf{a}, \mathbf{d}, \mathbf{s}^i, X) = \sum_{j=1}^{N_c} X_{ij} \times \left(a_j + \sum_{t=1}^{N_s} X_{tj} \times d_t \right) \quad (8)$$

State migration cost term M_{upg}^i : Excluding the updating states (e.g., s^i), other states in the same state sub-contract need to be migrated to the new version. For each state sub-contract containing s^i , we search the states (in addition to s^i) in it and calculate the migration cost of these states. The migration cost of each state is denoted as $\mathbf{m} = [m_1, \dots, m_{N_s}]$. For the state with dynamic storage layout, the corresponding m is adjusted based on its storage size specified in the provided DSL-written contract. Thus, the total migration cost of updating the state s^i in Equation (6) is expressed as follows:

$$\text{MigrateCost}(s^i, \mathbf{m}, X) = \sum_{j=1}^{N_c} \sum_{k=1}^{N_s} X_{ij} \times X_{kj} \times m_k, k \neq i \quad (9)$$

Besides the above three optimization terms, the storage affinity constraints must be satisfied in the hyperproxy-based contract maintenance pattern to ensure the correctness of the modularization process. The states involved in the same function of a logic contract should be grouped into the same state sub-contract. Additionally, states that exhibit interdependence (i.e., one state is directly derived from another) should also be grouped together. Building on those affinity features, we introduce an affinity matrix ($A_{N_s \times N_c}$) where all its entries are either 0 or 1. If states satisfy the aforementioned criteria, the corresponding element in $A_{N_s \times N_c}$ is 1, showing a high affinity between them. Accordingly, X should fulfill a requirement that the states with high affinity must be modularized into the same state sub-contract, as shown in Equation (2) where \odot is Hadamard (element-wise) product and $T \in \mathbf{0}_{N_s \times N_s}$. To ensure that each state is only allocated to one state sub-contract, X should also satisfy the constraint in Equation (3) where $C_1 \in \mathbf{1}_{N_c \times 1}$ and $C_2 \in \mathbf{1}_{N_s \times 1}$.

We formulate the optimization problem as a mixed-integer program (MIP) problem [21] to derive an optimal allocation matrix $X_{N_s \times N_c}$, which determines the allocation of states across state sub-contracts following the optimal contract modularization algorithm. Note that if developers do not specify storage sizes, SmartUpdater defaults to consolidating all states into a single state contract to minimize the deployment gas costs.

Contract Generation and Deployment. In the initial contract deployment phase, SmartUpdater parses the DSL-written contract through static analysis, extracting essential information needed for the optimal contract modularization algorithm. To achieve this, SmartUpdater parses the contract's abstract syntax tree (AST) to obtain the fundamental attributes of each state variable. Moreover, it leverages the static analyzer Slither [23] to extract precise relational information. Specifically, SmartUpdater traverses all state variables by inspecting "VariableDeclaration" nodes in the AST, determining their types from the "typeName" field. For states with deterministic storage layout, especially for the fixed-size array-type, SmartUpdater obtains their storage sizes according to the "length" field. For the state with dynamic storage layout (i.e., dynamic array-type and mapping-type), SmartUpdater identifies the annotation (i.e., @mag and @bat) in the DSL-written contract to acquire its storage size provided by the developer. Moreover, SmartUpdater captures state update

probabilities specified by @prob annotations as hyperparameters in the modularization algorithm. To determine the relational information, SmartUpdater utilizes the Slither to transform the entire contract to its internal representation language (i.e., SlithIR) to retrieve which states each function accesses and uses static single assessment (SSA) to compute the interdependencies between states.

With these extracted details (i.e., state attributes, their interdependencies, and their association with functions), SmartUpdater executes the optimal contract modularization algorithm and determines contract modularization strategies. Based on this strategy, SmartUpdater then generates pairs of sub-contracts, each consisting of a state sub-contract and a corresponding logic sub-contract that includes the functions interacting with the states in its paired state sub-contract. Finally, all sub-contracts are coordinated by the hyperproxy as introduced in Sec. IV-C.

In the maintenance process, the developer provides the DSL-written policies for SmartUpdater to precisely modify the state or logic. SmartUpdater first determines the policy type by analyzing the identifier @LogicPolicy or @StatePolicy, and retrieves the target contract address associated with this policy. In the logic modification policy, SmartUpdater obtains the revised contract code. In the state modification policy, SmartUpdater analyzes each operation statement to extract the action keyword and four tuples (i.e., identifier, type, value, and modifier) of the target state, obtaining the specific state update requirements. Based on this information, SmartUpdater generates the corresponding new sub-contracts. After deploying these new sub-contracts, the target contract address stored in the hyperproxy contract or relevant state sub-contracts is updated to the new versions.

F. Publicly Verifiable State Migration

State contract maintenance usually needs to restore states from the old state contract (i.e., state migration). Although all information is public on the blockchain system, not all states in one contract can be easily retrieved due to two reasons: 1) EVM state modifier mechanisms [2] make states that are declared as *private* and *internal* inaccessible for other external contracts; 2) states of different types may not be retrieved merely via state names. For example, keys are requisite to retrieve mapping-type states, as described in Sec. II. Yet, they are usually determined by user inputs and not stored on the chain.

To ensure correct state migration, we devise two components: 1) a hybrid state exposure method that customizes various retrieval interfaces according to state characteristics, such that states of different types can be retrieved on-chain; 2) a publicly verifiable state recovery method that employs an independent updater contract, which is assisted by a helper committee using a SNARK-based mechanism, to correctly retrieve states of arbitrary types from the blockchain system and initialize them in the newly deployed contracts. We emphasize that compared with prior call-based or delegatecall-based maintenance patterns [14], [34], which do not address the challenges of handling diverse states during migration, SmartUpdater

significantly enhances state migration capabilities by supporting more complex state types, such as mapping-type states.

Hybrid State Exposure Method. According to different state characteristics, `SmartUpdater` applies different exposure methods to ensure states are accessible on-chain. Usually, states with deterministic storage layout can be retrieved directly using state names. In particular, `SmartUpdater` generates customized *getter* functions for *private* and *internal* states as retrieval interfaces, such that the updater contract can recover the state values after the corresponding new contracts are deployed. Note that, to preserve the secure property of the *private* and *internal* states, these customized interfaces only allow an authorized address (i.e., the address of the updater contract) to access.

For states with dynamic storage layout, including dynamic array-type and mapping-type (described in Section II), `SmartUpdater` provides additional retrieval methods. For dynamic array-type states, `SmartUpdater` offers a function interface to return its length, which is essential for retrieving all values in the array. For mapping-type states, `SmartUpdater` introduces a shadow on-chain record mechanism for off-chain key inputs, such that all values of mapping-type states can be retrieved according to the shadow key records. These shadow key records are stored in the blockchain logs during contract execution. Compared with native smart contract storage, logs reside in transaction receipts of the blockchain (see Fig. 3) and, more importantly, are usually used as a means of *cheaper data storage* [31], [35].

`SmartUpdater` scrutinizes logic contracts and enables the log-based shadow records for each mapping-type state. Since EVM's logging operations are enabled via events, `SmartUpdater` declares an event with a specific topic identifier (denoted as t_m) for each mapping-type state. Then, it inserts an event emitting operation (encoding the corresponding key) for each write of a mapping-type state in the logic contract. Therefore, once a new element belonging to a mapping variable is inserted into the contract storage at runtime, an event with the key is emitted which is stored in the logs. These processes are automated during contract execution, ensuring the integrity of these shadow keys and facilitating verifiable state recovery (presented in the next subsection). Finally, these logs can be efficiently collected by a bloom filter in block headers for mapping-type state recovery, which will be described next.

Publicly Verifiable State Recovery. The updater contract, assisted by a helper committee using SNARK-based mechanism, is responsible for retrieving states of various types from the previous state contract and initializing the associated states in the newly patched contract. The correctness of all operations, including the state retrieval and initialization, can be verified by anyone independently.

Migrating non-mapping states: As illustrated in Algo. 1, the updater contract uses the `GeneralMig` function to retrieve states from the old state sub-contract through *getter* functions (generated by `SmartUpdater`) with corresponding state identifiers (see lines 4 in Algo. 1). The retrieved states are then initialized in the new state contract according to the state identifiers and types (see line 10 in Algo. 1).

Algorithm 1: The updater contract

```

1 Key :=  $\emptyset$ ;                                ▷ keys of the mapping-type state
2 isVerified :=  $\perp$ ;
3 procedure
  GeneralMig(addrO, addrN, staOld, staNew) :
4   valOld  $\leftarrow$  addrO.staOld.getter();
5   if staOld.type  $\neq$  staNew.type then
6     | valNew  $\leftarrow$  typeconversion(valOld);
7   else
8     | valNew  $\leftarrow$  valOld;
9   end
10  addrN.Initial(staNew, valNew);
11 end
12 procedure KeysVerify([p], [ $\pi$ ]) :
13   if each  $\pi$  is verified then
14     | Update Key according to the  $\{k^m\}$ ;
15     | isVerified  $\leftarrow$  true;
16   end
17 end
18 procedure
  MappingMig(addrO, addrN, staOld, staNew) :
19   if isVerified = true then
20     for each  $k \in$  Key do
21       | valOld  $\leftarrow$  addrO.staOld.getter( $k$ );
22       | if staOld.type  $\neq$  staNew.type then
23         | | valNew  $\leftarrow$  typeconversion(valOld);
24       | else
25         | | valNew  $\leftarrow$  valOld;
26       | end
27       | addrN.Initial(staNew,  $k$ , valNew);
28     end
29   else
30     | return  $\perp$                                 ▷ tell the caller to wait
31   end
32 end

```

Migrating mapping states: There are two stages in migrating mapping-type states: SNARK-based key collection that triggers a helper committee to obtain all shadow records for keys along with a SNARK proof for integrity guarantees; SNARK-based mapping-type state recovery that restores mapping-type states with desirable public verifiability. The details are as follows:

SNARK-based key collection. Nodes in a helper committee retrieve all keys for relevant mapping variables from the blockchain system and produce a SNARK proof showing the integrity of the retrieved results. Take as an example one mapping variable called $m[\cdot]$ including a set of keys denoted as $[k^m]$. The events related to $m[\cdot]$ are identified by a topic t_m , which is contained in the log (see Section II). All of them reside in the blockchain starting from the block where the contract is deployed (denoted as blk_s) to the block where the contract is locked (denoted as blk_e). The contract is locked once developers provide the logic/state modification policy, which prevents any changes in the state value during the contract maintenance.

Following the log searching operations in the blockchain as shown in Fig. 3, the nodes in the committee achieve *SNARK-based key collection* via five steps: a) The node obtains all block headers between blk_s and blk_e from blockchain full nodes. For each block header, the node generates a proof to prove that it is the correct one which sequentially follows its former. b) The node filters out these block headers by explicitly checking the *logBloom* field of each block header and creates a proof to show whether it contains the log with topic t_m . c) For each remaining block header, the node downloads its all relevant receipts and creates proofs of integrity. d) To further improve efficiency, the node filters each of these receipts while generating proofs. e) The node traverses logs of each unfiltered receipt, retrieves the keys identified by the topic t_m , and generates a proof as well.

SNARK-based mapping-type state recovery. Both the `KeysVerify` and `MappingMig` functions in the updater contract are used to migrate mapping-type states. `KeysVerify` is invoked by the helper committee to provide keys (denoted as $[k^m]$) and relevant proofs (denoted as $[\pi]$) for a certain mapping state. $[k^m]$ is in the verification parameters $[p]$ which includes the input and the output of each step. If $[\pi]$ are verified, it indicates that the obtained $[k^m]$ contains all keys of the target mapping-type state, and the updater contract then updates and maintains these keys (see lines 15-17 in Algo. 1). Since the caller of this function is required to pay a fee, DoS attacks are prevented.

The `MappingMig` function is called by the developer to migrate mapping-type states. It retrieves values of mapping-type states from the old state sub-contract based on its identifier and verified key (see lines 23-24 in Algo. 1). Subsequently, the updater contract calls the initial function of the new state contract to initialize the value of the associated key (see line 30 in Algo. 1).

In practice, `SmartUpdater` employs a batching migration strategy within the updater contract, aiming to reduce the gas costs incurred by migrating state with large storage sizes like array-type and mapping-type states. Specifically, `SmartUpdater` is configured to perform the migration of multiple state elements in a single transaction, rather than migrating each element individually. To ascertain the most effective batch size, we have conducted a detailed evaluation of the gas consumption. The findings are presented in Sec. V-B.

SNARK Proof Refinement. The most computationally demanding part of the state migration module is the SNARK-based proofs generation that the helper committee must do for every block between the contract deployment block and the contract locking block. Moreover, since Ethereum's average block time is 12s [20] and each block involves hundreds of receipts, retrieving keys of every mapping-type state requires a substantial number of proofs. Therefore, the major source of overhead in the state migration is the verification costs in the updater contract. To make the migration of mapping states practical, our model proposes two ideas:

Proof generation time reduction with hash guarantee. We observe that some of the circuits for *SNARK-based key collection* are massive, especially in the stage of obtaining all

block headers and obtaining relevant receipts, causing a huge amount of time on proving. To address the problem, one key observation is that hash functions are widely used in block constructions in blockchain. Therefore, the security properties of the hash function, our module can directly verify whether the correct preimage for a given Keccak-256 hash value is provided, instead of proving the whole computation like constructing the entire Merkle Patricia Tree. Take obtaining receipts and proving their integrity in one block as an example. Since the property of *preimage resistance* of the hash function, the helper committee cannot get the correct input based on the known hash value. Conversely, due to the *collision resistance* of the hash function, it is guaranteed that the receipt in $[r_i]$ has not been maliciously altered.

Smaller proof size with recursive verification. While verifying proofs on ordinary CPUs is very efficient, on-chain verification is still costly, especially for large proof size. To further reduce the on-chain verification cost (computation and storage), our module uses a 2-layer proof approach. The helper committee invokes *SNARK-based key collection* to generate proofs in the first layer. In the second layer, we use *recursive verification*: the prover demonstrates that the previous proofs from the first layer correctly validate the corresponding keys of a mapping-type state. At a high level, we trade increased proof generation time off-chain for much reduced on-chain cost: the required computation reduces from an infeasible amount of gas to 1024K gas which is an acceptable verification cost.

G. Implementation

We implemented `SmartUpdater` featured with the hyperproxy-based contract maintenance pattern, the expressive programming model, the optimization-oriented contract generation module, and the publicly verifiable state migration module. We developed a *full-fledged compiler* with the following functions: 1) modularizing the DSL-written smart contracts into sub-contracts based on MIP-based optimization and exposing various states of sub-contracts; 2) generating a hyperproxy contract and making it coordinate all sub-contracts; 3) translating the DSL-written policies into corresponding new sub-contracts; 4) generating updater contracts according to the policies. To implement all the functions, `SmartUpdater` first uses the Solidity compiler [41] to compile the DSL-written contract, ensuring the contract is free from compilation errors. Moreover, `SmartUpdater` generates the contract's abstract syntax tree (AST) through the Solidity compiler and utilizes a state-of-the-art static analyzer Slither [23] to further dissect the DSL-written contract, as described in Section IV-E. Furthermore, we solved the optimal contract modularization algorithm by using Gurobi Optimizer [27] (implemented in 300+ lines).

In addition to the full-fledged compiler, `SmartUpdater` implements SNARK proof circuits in the state migration module, meticulously designed using Circom [28]. The first layer of proofs can be implemented with any protocol (implemented circuits in 800+ lines) and we opted for Groth16 [26] as our second-layer SNARK protocol. This choice is motivated by

its advantages: constant proof size, rapid verification times, and compatibility with the BN254 curve natively supported by Ethereum, optimizing on-chain costs. We implemented a recursive verification circuit based on an open sourced verification algorithm [10] to check the validity of the proofs in the first layer. For each circuit, we assume that there will be a trustworthy third party to perform the trusted setup phase. We envision a future where the helper committee, pivotal to SmartUpdater's verifiable state migration, operates as a ubiquitous service within the network. To simulate this, we evaluated the SNARK-driven state migration in a controlled environment.

V. EVALUATIONS

In this section, we evaluate the overall performance of SmartUpdater with its hyperproxy-based maintenance pattern and the practicability of its key modules. The evaluation goals are as follows:

- Goal 1: Assess whether SmartUpdater consumes smaller gas costs for real-world maintenance cases throughout the contract maintenance process, compared to actual contract maintenance approaches (see Section V-B).
- Goal 2: Evaluate whether the optimization-oriented contract generation module reduces gas costs in various scenarios involving large-scale state migration (see Section V-C).
- Goal 3: Examine whether the publicly verifiable state migration module incurs acceptable proof sizes and on-chain proof verification costs under varied state migration scenarios (see Section V-D).
- Goal 4: Assess whether SmartUpdater's development interfaces streamline development processes for real-world smart contract maintenance cases, compared to existing contract development tools (see Section V-E).

A. Evaluation Setup

Datasets. We constructed two datasets to reflect (1) real-world contract maintenance cases and (2) extreme conditions with gas-cost-intensive maintenance involving large-scale state migrations. In the first dataset, we collected 149,164 real-world smart contracts based on XBlock-ETH [58] which is widely used and provides contract information of the Ethereum system. Among these, we identified 4,621 contract maintenance cases by grouping contracts with the same name and creator and arranging them chronologically. After analyzing their source code and identifying state changes (i.e., state appends and storage layout changes), we selected 1,355 maintenance cases as the default dataset. Each case includes the original contract, its new version, and their differences (e.g., varied state variable declarations). Moreover, to assess SmartUpdater's utility under extreme conditions, we built specific maintenance cases involving massive state migrations. Using Etherscan [22], we ranked ERC-20 contracts by the number of holders, which correlates with the number of mapping-type state elements. We selected four representative ERC-20 contracts with the highest number of mapping elements. We chose the common states (i.e., *name*,

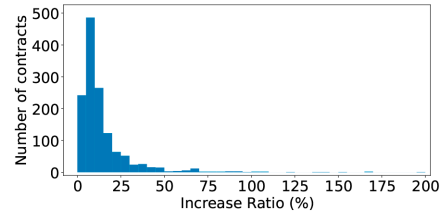


Fig. 8. The distribution of contract numbers with the rate of code size increases.

decimals, *totalSupply*, and *balance*) from these ERC-20 contracts as the targets for updates, to simulate scenarios involving large-scale state migrations.

Experimental Setting. We used ECS G7A 16xlarge instances equipped with the Intel(R) Xeon(R) Platinum 8275CL CPU@3.00GHz and 256GB of RAM to conduct the experiments. To offer an extensive perspective, we document both the resource and monetary costs associated with our evaluations. Following the threat assumption in Sec. IV, we assume one honest node in the helper committee. We evaluate the performance of SmartUpdater in depth based on the maintenance conditions in those contracts. We set the hyperparameter \mathbf{p} as $[1, \dots, 1]$ in the optimal contract modularization algorithm to treat all states with the same updating probabilities for simplicity. All experimental results are averaged after 10 independent trials.

B. Overall Gas Consumption of SmartUpdater

We extensively evaluated the gas consumption caused by SmartUpdater in comparison with actual contract maintenance approaches when handling real-world contract maintenance cases based on the default dataset. This evaluation focuses on state updates and encompasses both the initial deployment and contract maintenance process. Our analysis of the default dataset indicates the actual contract maintenance approaches employed in real-world contracts. These methods include renewing contracts and leveraging the call-based maintenance pattern. The delegatecall-based pattern, which does not support storage layout changes, was not utilized in any of the cases within the dataset. Due to the unavailability of data regarding the storage size for states with dynamic storage layout (e.g., mapping-type states), we standardize the size to 10 for evaluation (see Sec. V-C).

Gas Consumption in Initial Deployment. We evaluate the additional gas consumption when using SmartUpdater for each original contract in the maintenance cases. Because deployment costs in the Ethereum system are proportional to the size of the deployed contract [53], understanding the factors of additional code size inflation is critical. In SmartUpdater, inflation primarily arises from the integration of a hyperproxy contract, shadow-record emitting events, and some duplicated code like delegatecall-based functions. The inflation size, expressed as a percentage of the original code size, is termed the increase ratio. Fig. 8 shows the distribution of contracts across various increase ratio ranges, divided into 5% intervals. The results

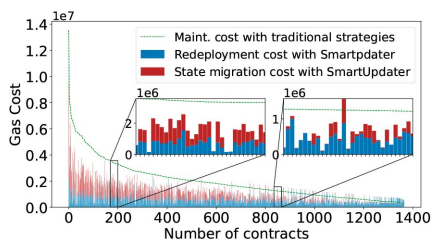


Fig. 9. Comparison of maintenance costs with and without using SmartUpdater.

indicate that the most common increase ratio falls within the 5% to 10% range, demonstrating that our proposed pattern introduces relatively minimal code. Among these, the smallest observed increase in gas consumption was 126,600 gas (\$13.9). The average increase ratio is 15%, equating to a gas consumption of 169,331 (\$18.7), which is considered acceptable given the security enhancement provided by the pattern. Conversely, approximately 3% of contracts exhibit significant code size increases, exceeding 50%. In these cases, the highest observed increase ratio was 196%, resulting in a maximum increase in gas consumption of 494,600 gas (\$54.6). These contracts are primarily smaller in size and have fewer states. As a result, the additional pattern features lead to a more substantial relative increase in gas cost.

The size of shadow-record emitting events is determined by its parameters. However, many token standards, like ERC-20³ and ERC-223⁴, require the inclusion of events to log information that includes the keys of the target mapping-type state. This means that it is not necessary to introduce additional duplicate events for those states. In our analysis of thirty ERC-20 tokens with the highest number of holders, we identified 80 mapping variables, of which only 4 mappings are not followed by the corresponding event to record the key. This reinforces the validity of using the event to record keys in mapping-type states.

Gas Consumption in Contract Maintenance. We evaluated the gas consumption of contract maintenance, which includes deployment costs of new contracts and state migration costs from the old version. We further compare the gas costs of contract maintenance between the actual approaches used in the maintenance cases and SmartUpdater. As shown in Fig. 9, SmartUpdater demonstrates significant improvements in gas-cost efficiency. Specifically, 94% of contracts experience lower gas costs when using SmartUpdater compared to the usage of their actual maintenance approach, resulting in average savings of 1.2 million gas (\$132) per contract. These savings stem from SmartUpdater’s optimization-oriented contract generation module, which enables the division of the contract to avoid the migration of states that are not pertinent to the specific state update. However, 6% of the contracts experience increased gas consumption. This was typically due to the high affinity of state variables within these contracts, which hindered effective modularization. Moreover, using SmartUpdater introduces a slight

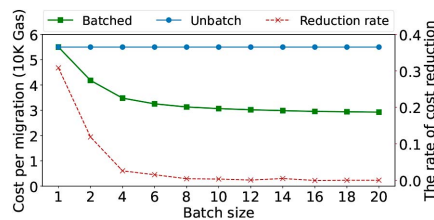


Fig. 10. Gas cost with varying batch sizes.

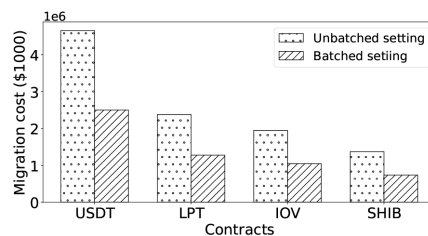


Fig. 11. Migration cost of mapping-type state in selected contracts.

overhead due to the configuration of the new contract address, yet the overall increase is marginal, approximately \$0.6.

We also evaluate the gas costs of the batching migration strategy for states with large storage sizes like array-type and mapping-type. As depicted in Fig. 10, the unbatched approach incurs a consistent gas cost of approximately 55000 gas (\$5.7), whereas the batched approach displays a descending trend in costs as the batch size within the contract expands. We observe that when the batch size is 16 or greater, the rate at which cost reductions are achieved begins to decelerate. In other words, increasing the batch size yields diminishing returns in terms of cost efficiency at this point. Considering additional factors such as transaction latency and block gas limits, as well as the observed deceleration in cost benefits, we opt for a batch size of 16 as a reasonable trade-off. We also evaluated the mapping-type state migration cost of prevalent ERC-20 contracts with the heaviest mapping elements in Fig. 11 and found that the use of the batching strategy saves about 50% gas costs. This finding highlights the performance of the batching technique, especially for extensive state migrations.

Furthermore, we investigate the gas consumption throughout the whole contract maintenance process under varied real-world state update scenarios. In our default dataset, 27% of these real-world contract maintenance cases only append states without completely modifying the storage layout, while others change the storage layout, e.g., via removing states or changing the state type. In the scenario of simple state appends, 56% cases save gas consumption throughout the whole contract maintenance process when using SmartUpdater, compared to the actual contract maintenance approach. The average savings is 630,105 gas (\$69.6). In the scenario of storage layout modifications, 78% of those cases experienced lower gas costs with SmartUpdater, resulting in an average reduction of 731,256 gas (\$80.8). Other cases incur more gas costs (186,213 gas on average, approximately \$20.6) mainly due to the code size inflation of using hyperproxy-based patterns. In summary, 72%

³<https://eips.ethereum.org/EIPS/eip-20>

⁴<https://eips.ethereum.org/EIPS/eip-223>

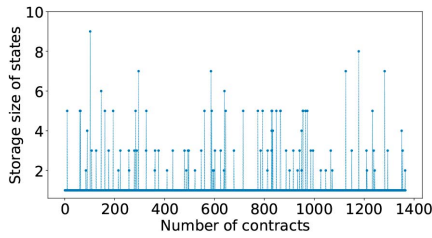


Fig. 12. Storage size of mapping-type state which causes the different modularization results.

of all real-world contract maintenance cases in our default dataset demonstrate gas-cost-efficiency with SmartUpdater, achieving an average saving of 711K gas (\$78.5).

C. Evaluation of Optimization-Oriented Contract Generation

In this section, we leverage our datasets with determined update states to evaluate the gas cost efficiency of SmartUpdater's optimization-oriented contract generation module. Firstly, we confirm that when using the modularization algorithm of this module, the storage attribute predefined by developers exerts little influence on the optimization accuracy. Subsequently, we use our datasets to compare three modularization algorithms: 1) MIP-based contract modularization algorithm (i.e., the optimal contract modularization algorithm we provided in Section IV-E); 2) Minimum-state modularization algorithm that assigns each state sub-contract with states involved in a single function; 3) Maximum-state modularization algorithm that makes the state contract obtain all states. The latter two modularization schemes commonly serve as benchmarks in various research domains. Our analysis then focuses on specific contract maintenance cases that incur massive state migrations, evaluating whether the optimization-oriented contract generation module still performs gas-cost-efficiency in extreme cases.

Evaluation Under Different Storage Attributes. In this part, we analyze the impact of optimization accuracy of contract modularization with different storage attributes that the developers set in the programming model. This evaluation involves varying the storage size of the states with dynamic storage layout to observe changes in the modularization results. The results, as depicted in Fig. 12, indicate that the majority of contracts do not show any variation and the number of affected contracts is relatively small. Specifically, when the storage size is changed, only 85 contracts exhibit different modularization outcomes. Notably, these changes occur within a narrow range of storage sizes, with all critical transitions occurring at storage sizes below 10. Therefore, the choice of storage size has a limited impact on the optimization accuracy of contract modularization. However, it is also advisable for developers to configure their contracts with larger storage sizes based on actual needs.

Evaluation On Large-scale Real-world Contracts. In this part, we analyze the maintenance gas costs among different modularization algorithms mentioned above. The results are shown in Fig. 13 with the costs normalized to those of the MIP-based modularization algorithm. Note that since the storage

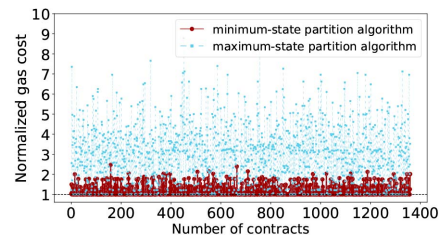


Fig. 13. On-chain costs with various optimization algorithms normalized to the MIP-based modularization algorithm.

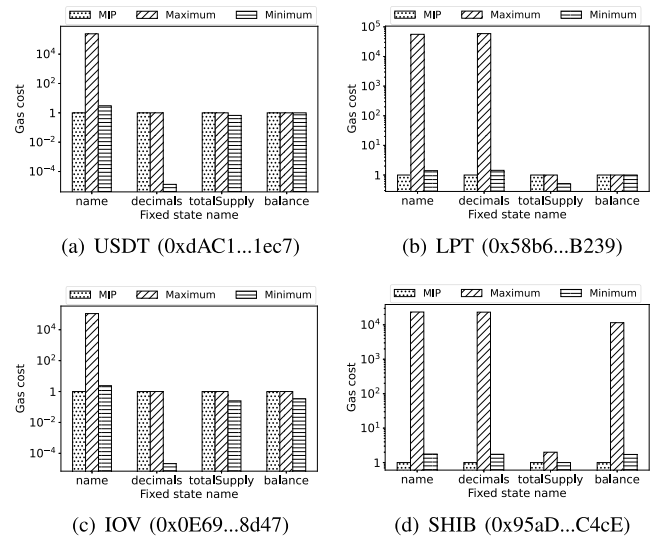


Fig. 14. On-chain costs of notable states in various contracts.

size of each state with dynamic storage layout (e.g., mapping-type state) is not accessible, we assume the quantity to be 10 to better present the comparison results. The MIP-based modularization algorithm outperforms the others, reducing the overall maintenance cost by up to 8.8 times. This suggests that the more balanced modularization generated by the MIP-based modularization algorithm can significantly save gas costs in contract maintenance. Besides, we found that the maximum-state modularization algorithm generally performs poorly. This is because all states are stored in a single state contract, leading to unnecessary migration of unrelated states during updates and incurring extra costs.

Evaluation Under Massive State Migration. We utilize the selected four ERC-20 tokens with the highest number of mapping elements to demonstrate that SmartUpdater's optimization-oriented contract generation module still yields optimal optimal contract modularization strategy and saves gas costs in massive state migration cases. The customized update states include *name*, *decimals*, *totalSupply*, and *balance*, with types *string*, *uint256*, *uint256*, and *mapping*. We compare the gas costs when using three different modularization algorithms to update each state (normalizing to the MIP-based modularization algorithm), and the evaluation results are reported in Fig. 14.

TABLE II
COMPLEXITY OF CORE TASKS

Verification Task	# of Constraints
Block header continuity	617070
Bloom filter	158995
Receipts matching	152240
Data retrieval	1536

Compared to the maximum-state modularization algorithm, it is evident that using the MIP-based modularization algorithm consistently reduces gas costs for any state updates. The reduction in gas costs can be as significant as four orders of magnitude. Besides, the benefits of our design for some state variables (like *decimals* in USDT) are not significant, because the migration of necessary yet heavy mapping-type states dominates the overall costs. In contrast, the minimum-state modularization algorithm has less gas costs than using the MIP-based modularization algorithm in some update cases (like *totalSupply* in LPT) due to no unnecessary state migration introduced. However, the minimum-state modularization algorithm needs to preserve state consistency across different state sub-contracts serving for different functions, incurring an unquantifiable cost.

D. Evaluation of Publicly Verifiable State Migration

In this section, we assess the practicality of the publicly verifiable state migration module focusing on proof sizes and on-chain proof verification costs, in comparison with unrefined SNARK protocols [28]. We selected real-world contracts from our default dataset that have migration requirements of the mapping-type state and follow the ERC-20 token standard, which already uses events to record the keys of the mapping-type state. We randomly choose three of them (i.e., XRT, ACNT, and IBTC) as examples for the evaluation of the real-world situation. There are only two mapping-type states, i.e., *allowed* and *transfer*, that are migrated in contract maintenance. We evaluate the costs for the *allowed* state with relatively smaller members, as it is enough to show that the original SNARK protocol already incurs prohibitive gas costs and our refined design achieves constant overhead.

Succinct Proof Size. To provide further insight into the proof generation costs, Table II tabulates the number of resultant rank-1 constraint system (R1CS) constraints for every operation in the SNARK-based key collection mechanism (see Section IV-F), which is proportional to the proof generation time. The most resource-intensive operation is the validation of header continuity, attributed to the intricate Keccak-256bit hash function with voluminous inputs. Additionally, the task of matching related receipts is slightly more efficient with the same hash function, due to its smaller input size. Filtering operations stand out in their efficiency, only necessitating an $O(1)$ complexity circuit. Furthermore, even if one single log's key retrieval is straightforward, the complexity escalates with multiple logs and up to three topics in each.

Table III shows the different proof sizes under non-recursive and recursive verification designs. For the ACNT contract

whose deployment and maintenance cross 20530 blocks, the proof size is reduced from 83 MB (across 21288 proofs) to 55 KB (spanning 5 proofs) when using the recursive refinement. Despite the longer generation time, the remarkable contraction in proof size—roughly a thousandfold—is undeniably significant.

On-chain Proof Verification Cost. Table III shows that the on-chain verification cost is constant (1024K) when using the recursive refinement, which is roughly \$105. If we select ACNT contract, for instance, the verification cost becomes \$0.005 per block. In contrast, without our refinement, straightforwardly verifying these generated proofs on-chain would be prohibitively expensive, costing more than \$450000. Additionally, for the selected four ERC-20 contracts with massive mapping elements, the blocks they span (over 10 million blocks) and the number of receipts they record (more than 20 million) far exceed those of the real-world contracts. As a result, the verification cost would be significantly higher without our refinement, highlighting the benefits of our public verifiable state migration module when dealing with the extreme scenario.

E. Developer Usability Study

In this section, we evaluate the effort required by developers when using SmartUpdater's development interfaces throughout the whole contract maintenance process in real-world cases, and compare it to the effort involved in manual implementation and OpenZeppelin [38].

Participant Recruitment. We conducted a thorough study with nine professional developers, all recruited through the computer science major of our university. Of them, two were undergraduates, four were graduate students, and three were Ph.D. candidates. The undergraduates had taken blockchain courses and conducted relevant experiments with smart contracts. Other participants had been involved in blockchain-related projects and four of them (one graduate student and three Ph.D. candidates) previously used the delegatecall-based pattern.

Study Methods. To ensure an exhaustive evaluation of diverse maintenance scenarios, we provided developers with ten contracts of varying complexity and maintenance requirements as study samples. Among these, three contracts were created by us as relatively simple examples, which include basic state (e.g., string or integer type) and straightforward logic (e.g., reading or writing state value). These contracts were designed to involve only logic update requirements. The remaining contracts were selected from our default database which consists of real-world maintenance cases. These contracts have larger code sizes and more sophisticated logic functionalities. More importantly, these contracts involve various state update requirements, such as adding additional states (without changing storage layouts) and modifying storage layouts.

We evaluated developer usability across three approaches: manual implementation, OpenZeppelin, and SmartUpdater. Developers were required to use these approaches to perform two tasks: 1) converting smart contracts to follow certain contract maintenance patterns; 2) implementing the new contracts based on the provided maintenance requirement and recovering

TABLE III
COMPUTATION AND STORAGE COSTS OF RECURSIVE VERIFICATION

Contract	# of blocks	# of receipts	Proof Size (KB)		Verification Cost (Gas)	
			w/o RV	w/ RV	w/o RV	w/ RV
XRT(0x9909...9568)	661888	1081	2651972	55	135784M	1024K
ACNT(0xedd4...b6fe)	20530	740	85152	55	4359M	1024K
IBTC(0xB7c4...7532)	2148051	67455	8867836	55	454048M	1024K

the unchanged states in the new contract. Since `SmartUpdater` generates contracts using the novel hyperproxy-based contract maintenance pattern and includes unique modules (e.g., optimization-oriented contract generation module), we assumed that developers would be unaware of these details when using baseline approaches. This assumption was made to prevent overly complex task demands for developers. Developers were required to produce the `delegatecall`-based patterns without the need to consider contract modularization. For the convenience of state migration, we provided the values of all states to be migrated, ensuring consistency across the three methods.

Result Collection. Following the evaluation methodologies from state-of-the-art contract maintenance research [44], we provided developers with a detailed study guide and structured questionnaires. The study guide included step-by-step instructions for task completion, environment setup, and submission guidelines. The questionnaires captured evaluation metrics, including task completion time, developer-perceived confidence level in task correctness, and developer-perceived task difficulty ratings. The latter two metrics were assessed using a 7-point Likert scale [30], with clear rating standards provided. Although developers completed tasks independently, the study guide ensured consistent procedures across participants. To validate the results, we required developers to submit all edited files (e.g., smart contracts created during the tasks) alongside their questionnaire responses. All results were anonymized to eliminate potential bias. We then conducted a manual code review of the submitted contracts and compared them with those generated by `SmartUpdater` and other approaches.

Results of Task 1. The results of task 1 showed that developers spent an average of 49.44 minutes on manual implementation and only two developers successfully converted. It was also reflected in a median difficulty level of 5 and a median confidence level of 3 in the correctness of performing the task. The main issue we observed was that developers overlooked the storage collision, resulting in the converted contracts being broken by design. Using `OpenZeppelin`, conversion time dropped to 35.22 minutes per contract, and six developers successfully converted contracts. Additionally, the median level of task difficulty and developers' confidence changed to 4, indicating that using `OpenZeppelin` facilitates contract development following the contract maintenance pattern. In comparison, `SmartUpdater` requires no prior knowledge about the contract contents, enabling developers to accurately convert contracts following the hyperproxy-based contract maintenance pattern within at most 6 minutes. Moreover, developers

generally agreed that using `SmartUpdater` strongly reduces the conversion difficulty, and expressed high satisfaction with the `SmartUpdater`, with a median satisfaction rating of 6. This confirms that using `SmartUpdater` decreases developer efforts to construct contracts with a maintenance pattern.

Results of Task 2. For contracts requiring only logic updates or simple state appends, all developers successfully complete the task in under 7 minutes, with minimal differences across using three methods. For contracts with state update requirements (i.e., storage layout changes), developers required an average of 32.22 minutes with manual implementation, while using `OpenZeppelin` reduced to 28.89 minutes. Particularly, most developers encountered storage collision issues during maintenance, with only two developers successfully identifying and resolving the problem. When asked about retrieving mapping-type states and their associated challenges, only three developers recognized the importance of recording the keys. However, their proposed solutions have shortcomings: 1) storing the keys of mapping-type states in contract storage would increase resource consumption [31] and result in high gas costs. 2) storing the mapping-type state in an independent contract would add contract complexity and incur additional gas costs during inter-contract calls. These findings indicate a general lack of EVM expertise of developers, as well as a limited awareness of how to preserve data integrity in contract maintenance, particularly for mapping-type states. In contrast, `SmartUpdater` streamlines the developers' work in the contract maintenance process, especially involving the storage layout changes. On average, developers completed the maintenance tasks correctly in just 5.22 minutes. Moreover, developers considered contract maintenance with `SmartUpdater` to be easier (median difficulty level of 2) than with the other two methods, which had a median difficulty level exceeding 4.

VI. DISCUSSION

In this section, we present the principal findings, contributions to both practice and research, as well as limitations and further directions.

Principal Findings. `SmartUpdater` achieves user-transparent contract maintenance by introducing the hyperproxy-based pattern. Although the hyperproxy contract increases initial deployment gas costs, this overhead is effectively offset by the gas-cost-efficient contract maintenance design enabled by the optimization-oriented contract generation module. Our study shows that while the initial deployment experiences an

average 15% gas increase, SmartUpdater significantly reduces gas consumption during contract maintenance compared to the actual maintenance approaches, making it particularly advantageous for frequent contract maintenance.

Contributions To Practice. SmartUpdater allows developers to address diverse contract maintenance requirements, including complex storage layout changes, via its domain-specific language (DSL). By automating optimization-oriented contract generation and correctness-guaranteed state migration, SmartUpdater decreases manual effort and developer errors. Besides, SmartUpdater can integrate with patterns proposed by Ethereum Improvement Proposals (EIPs), addressing delegatecall-based maintenance concerns such as storage layout compatibility [4], [5], function selector collisions [3], [9], and atomic maintenance [6]. For instance, the state sub-contract in our hyperproxy-based contract maintenance pattern can leverage a designated slot to record the corresponding logic sub-contract address, ensuring storage layout compatibility. Moreover, SmartUpdater can allow developers to define function groups within the DSL-written contract, enabling logical separation and facilitating atomic maintenance.

Contributions To Research. SmartUpdater proposes a hyperproxy-based contract maintenance pattern that first preserves user-transparent access while enabling any state and logic updates, contributing to smart contract design. At the same time, by leveraging advanced techniques like SNARKs, SmartUpdater ensures correctness in complex state migrations with acceptable gas consumption, setting a foundation for secure and efficient contract maintenance in resource-expensive blockchain systems.

Limitations And Future Directions. We discuss the limitations of SmartUpdater and propose potential solutions for our future work in the following aspects:

Elimination of Potential Bias in Usability Study. We recruited developers from the university, which may introduce familiarity bias or social desirability bias. For instance, participants may have been inclined to present themselves in a more favorable light. To further eliminate biases, we will offer financial rewards and recruit a more diverse and larger group of developers from various developer forums in the future. To add more detailed records of the study process, we plan to incorporate video recordings of participants' interactions during task completion. This will allow us to compare written reports with real-time actions observed in the video. By doing so, we can determine whether the participants accurately recorded the evaluation metrics (e.g., developer-perceived task difficulty ratings).

Composition of Helper Committee. The helper committee can operate in various capacities (e.g., as full nodes or using infrastructure like Infura [7]) to retrieve data and generate proofs. Future work includes incentivizing participation through rewards for validated proofs, inspired by incentive models like [16], [56].

Testing of Contract Maintenance. SmartUpdater can take an extensive testing approach to validate contract updates, enhancing future security. It can use developer-customized unit tests and integrate specialized tools, like static and dynamic

analysis methods [18], [45] to detect vulnerabilities in the updated contracts. We leave those as future work.

Support of More Contract Languages. Smart contracts are being developed in various high-level languages like Solidity, Vyper, and Go, and run on various blockchains like Solana and Hyperledger Fabric. To make SmartUpdater universally applicable, it can incorporate a language-agnostic Intermediate Representation (IR). This IR serves as a unifying layer that standardizes the core functionalities of smart contracts, irrespective of the original programming language. Moreover, SmartUpdater can adapt state migration methods to the specific storage layout of different blockchains. Extending SmartUpdater to other languages or even other blockchain systems is an interesting future research.

VII. CONCLUSION

Smart contracts maintenance is a central challenge in blockchain technology. Many maintenance approaches exist, but none offers a transparent, automated, and secure toolchain for both logic and state updates. We present SmartUpdater that employs a hyperproxy-based contract maintenance pattern to keep the maintenance process transparent to end users. SmartUpdater proposes various techniques to make the entire maintenance pipeline development-streamlining, gas-cost-efficiency, and state-migration-verifiable. Our evaluation results showcase SmartUpdater's effectiveness and cost-efficiency using 1,355 real-world contract maintenance cases. Moreover, experimental results demonstrate SmartUpdater significantly speeds up the maintenance process and integrates smoothly into the developers' workflow.

REFERENCES

- [1] "Smartupdater." Accessed: Nov. 1, 2024. [Online]. Available: <https://github.com/Ellen-syq/SmartUpdater>
- [2] "Visibility in solidity smart contracts." Accessed: May 2023. [Online]. Available: <https://cryptomarketpool.com/visibility-in-solidity-smart-contracts/>.
- [3] EIP-1538, "Transparent contract standard," 2018. Accessed: Nov. 2024. [Online]. Available: <https://eips.ethereum.org/EIPS/eip-1538>
- [4] EIP-897, "The first real proxy," 2018. Accessed: Nov. 2024. [Online]. Available: <https://ethereum-blockchain-developer.com/110-upgrade-smart-contracts/07-eip-897-proxy/>
- [5] Eip-1967, "Proxy storage slots," 2019. Accessed: Nov. 2024. [Online]. Available: <https://eips.ethereum.org/EIPS/eip-1967>
- [6] "EIP-2535: Diamonds, multi-facet proxy," 2020. Accessed: Nov. 2024. [Online]. Available: <https://eips.ethereum.org/EIPS/eip-2535>
- [7] "Infura: Ethereum & IPFS API," 2021. Accessed: Sep. 2023. [Online]. Available: <https://infura.io/>
- [8] "Solidity documentation," 2021. Accessed: May 2023. [Online]. Available: <https://docs.soliditylang.org/en/latest/>
- [9] EIP-1822, "Universal upgradeable proxy standard (UUPS)," 2022. Accessed: Nov. 2024. [Online]. Available: <https://eips.ethereum.org/EIPS/eip-1822>
- [10] "0xPARC. circom-pairing." 2022. Accessed: Aug. 2023. [Online]. Available: <https://github.com/yi-sun/circom-pairing/>
- [11] M. Ayub, T. Saleem, M. Janjua, and T. Ahmad, "Storage state analysis and extraction of ethereum blockchain smart contracts," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 3, pp. 1–32, 2023.
- [12] R. Behnke, "Explained: The lendhub hack," 2023. Accessed: Oct. 2024. [Online]. Available: <https://www.halborn.com/blog/post/explained-the-lendhub-hack-january-2023>

- [13] R. Behnke, "Explained: The pike finance hack," 2024. Accessed: Oct. 2024. [Online]. Available: <https://www.halborn.com/blog/post/explained-the-pike-finance-hack-april-2024>
- [14] W. E. Bodell, III, S. Meisami, and Y. Duan, "Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains," in *Proc. USENIX Secur.*, 2023, pp. 1829–1846.
- [15] B. Bü, S. A. nz, M. Zamani, and D. Boneh, "Zether: Towards privacy in a smart contract world," in *Proc. FC*, 2020, pp. 423–443.
- [16] D. Chen, H. Yuan, S. Hu, Q. Wang, and C. Wang, "BOSSA: A decentralized system for proofs of data retrievability and replication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 4, pp. 786–798, Apr. 2021.
- [17] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 327–345, Jan. 2022.
- [18] T. Chen et al., "Soda: A generic online detection framework for smart contracts," in *Proc. NDSS*, 2020, pp. 1–17.
- [19] Y. Chen, Y. Wang, M. Goyal, J. Dong, Y. Feng, and I. D. İş, "Synthesis-powered optimization of smart contracts via data type refactoring," *Proc. ACM Program. Lang.*, vol. 6, pp. 560–588, Oct. 2022.
- [20] Corwintines, "Blocks," 2023. Accessed: Sep. 2023. [Online]. Available: <https://ethereum.org/en/developers/docs/blocks/>
- [21] E. Danna, E. Rothberg, and C. L. Pape, "Exploring relaxation induced neighborhoods to improve mip solutions," *Math. Program.*, vol. 102, pp. 71–90, Jan. 2005.
- [22] Ethereum, "Token tracker (ERC-20)," Accessed: Nov. 2024. [Online]. Available: <https://etherscan.io/tokens>
- [23] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM WETSEB*, 2019, pp. 8–15.
- [24] C. F. Torres, M. Baden, R. Norvill, B. B. F. Pontiveros, H. Jonker, and S. Mauw, "Egis: Shielding vulnerable smart contracts against attacks," in *Proc. ACM ASIACCS*, 2020, pp. 584–597.
- [25] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *Proc. USENIX Secur.*, 2020, pp. 2557–2774.
- [26] J. Groth, "On the size of pairing-based non-interactive arguments," in *Proc. EUROCRYPT*, 2016, pp. 305–326.
- [27] "Gurobi optimization." Gurobi. Accessed: Aug. 2023. [Online]. Available: <https://www.gurobi.com/>
- [28] "iden3," *Circom*. 2023. Accessed: Aug. 2023. [Online]. Available: <https://github.com/iden3/circom>, 2023.
- [29] H. Jin, Z. Wang, M. Wen, W. Dai, Y. Zhu, and D. Zou, "AROC: An automatic repair framework for on-chain smart contracts," *IEEE Trans. Softw. Eng.*, vol. 48, no. 11, pp. 4611–4629, Nov. 2022.
- [30] A. Joshi, S. Kale, S. Chandel, and D. Pal, "Likert scale: Explored and explained," *British J. Appl. Sci. Technol.*, vol. 7, no. 4, pp. 396–403, 2015.
- [31] "Challenges and common solutions in smart contract development," *IEEE Trans. Softw. Eng.*, vol. 48, no. 11, pp. 4291–4318, Nov. 2022.
- [32] Z. Li, B. Xiao, S. Guo, and Y. Yang, "Securing deployed smart contracts and defi with distributed TEE cluster," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 3, pp. 828–842, Mar. 2023.
- [33] S. Ma, Y. Deng, D. He, J. Zhang, and X. Xie, "An efficient NIZK scheme for privacy-preserving transactions over account-model blockchain," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 2, pp. 641–651, Mar./Apr. 2020.
- [34] "Upgrading smart contracts," MobinHajizadeh. 2023. Accessed: May 2023. [Online]. Available: <https://ethereum.org/en/developers/docs/smart-contracts/upgrading/>
- [35] "Ethereum logs and events – what are event logs on the Ethereum network?" Moralis. Accessed: Aug. 2023. [Online]. Available: <https://moralis.io/ethereum-logs-and-events-what-are-event-logs-on-the-ethereum-network>
- [36] T. Bits, "How contract migration works." Accessed: May 2023. [Online]. Available: <https://blog.trailofbits.com/2018/10/29/how-contract-migration-works/>, 2018.
- [37] "Opensea nft marketplace," OpenSea. Accessed: Apr. 2023. [Online]. Available: <https://opensea.io/>, 2022.
- [38] openzeppelin, "Upgrades plugins." Accessed: Nov. 2024. [Online]. Available: <https://docs.openzeppelin.com/upgrades-plugins>
- [39] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," *Commun. ACM*, vol. 59, no. 2, pp. 103–112, 2016.
- [40] "President and fellows of harvard college," 2007. Extended Backus Naur Form (EBNF), Accessed: Nov. 2024. [Online]. Available: <https://www.arp.harvard.edu/eng/das/manuals/EBNF.html>
- [41] py-solc x., "Using the compiler," Accessed: Aug. 2023. [Online]. Available: <https://solcx.readthedocs.io/en/latest/using-the-compiler.html>, 2020.
- [42] A. Razaq, J. Buckley, Q. Lai, T. Yu, and G. Botterweck, "A systematic literature review on the influence of enhanced developer experience on developers' productivity: Factors, practices, and recommendations," *ACM Comput. Surveys*, vol. 57, no. 1, pp. 1–46, 2024.
- [43] Virtue Market Research, "Decentralized application development (DAPPS) market size," 2023. Accessed: Apr. 2023. [Online]. Available: [https://virtuemarketresearch.com/report/decentralized-application-development-\(dapps\)-market](https://virtuemarketresearch.com/report/decentralized-application-development-(dapps)-market)
- [44] M. Rodler, W. Li, G. O. Karame, and L. Davi, "EVMPatch: Timely and automated patching of ethereum smart contracts," in *Proc. USENIX Secur.*, 2021, pp. 1289–1306.
- [45] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VERISMART: A highly precise safety verifier for Ethereum smart contracts," in *Proc. IEEE SP*, 2020, pp. 1678–1694.
- [46] S. Steffen, B. Bichsel, R. Baumgartner, and M. Vechev, "ZeeStar: Private smart contracts by homomorphic encryption and zero-knowledge proofs," in *Proc. IEEE SP*, 2022, pp. 179–197.
- [47] S. Steffen, B. Bichsel, and M. Vechev, "Zapper: Smart contracts with data and identity privacy," in *Proc. ACM CCS*, 2022, pp. 2735–2749.
- [48] P. Tolmach, Y. Li, and S.-W. Lin, "Property-based automated repair of defi protocols," in *Proc. ASE*, 2022, pp. 1–5.
- [49] P. Tsankov, A. Dan, D. Drachler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM CCS*, 2018, pp. 67–82.
- [50] Y. Wang, X. Chen, Y. Huang, H.-N. Zhu, J. Bian, and Z. Zheng, "An empirical study on real bug fixes from solidity smart contract projects," *J. Syst. Softw.*, vol. 204, no. 111787, 2023, Art. no. 111787.
- [51] Y. Wang, J. Dong, R. Shah, and I. Dillig, "Synthesizing database programs for schema refactoring," in *Proc. ACM PLDI*, 2019.
- [52] Z. Wang, H. Jin, W. Dai, K.-K. R. Choo, and D. Zou, "Ethereum smart contract security research: Survey and future research opportunities," *Frontiers Comput. Sci.*, vol. 15, pp. 1–18, Apr. 2021.
- [53] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, no. 2014, pp. 1–32, 2014.
- [54] G. Wu, H. Wang, X. Lai, M. Wang, D. He, and S. Chan, "A comprehensive survey of smart contract security: State of the art and research directions," *J. Netw. Comput. Appl.*, vol. 226, 2024, Art. no. 103882.
- [55] T. Xie et al., "zkBridge: Trustless cross-chain bridges made practical," in *Proc. ACM CCS*, 2022, pp. 3003–3017.
- [56] C. Ying, H. Jin, J. Li, X. Si, and Y. Luo, "Incentive mechanism design via smart contract in blockchain-based edge-assisted crowdsensing," *Frontiers Comput. Sci.*, vol. 19, no. 3, 2025, Art. no. 193802.
- [57] X. Zhang et al., "EVM-shield: In-contract state access control for fast vulnerability detection and prevention," *IEEE Trans. Inf. Forensics Security*, vol. 19, pp. 2517–2532, 2024.
- [58] P. Zheng, Z. Zheng, J. Wu, and H.-N. Dai, "XBLOCK-ETH: Extracting and exploring blockchain data from Ethereum," *IEEE Open J. Comput. Soc.*, vol. 1, pp. 95–106, 2020.
- [59] H. Zhu, L. Yang, L. Wang, and V. S. Sheng, "A survey on security analysis methods of smart contracts," *IEEE Trans. Services Comput.*, vol. 17, no. 6, pp. 4522–4539, Nov./Dec. 2024.
- [60] W. Zou et al., "Smart contract development: Challenges and opportunities," *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2084–2106, Oct. 2019.

Xiaoli Zhang received the Ph.D. degree from Tsinghua University, China, under the supervision of Prof. Jianping Wu, in 2020. Currently, she is an Associate Professor with the School of Computer & Communication Engineering, University of Science and Technology Beijing, China. Her research interests include trusted computing, verifiable computation, cloud security, and network security.

Yiqiao Song is currently working toward the M.E. degree with Zhejiang University of Technology, Hangzhou, China. Her research interests include blockchain, cryptography, and information security.

Yuefeng Du received the B.S. (First Class Hons.) and Ph.D. degrees from the City University of Hong Kong, in 2018 and 2022, respectively. Currently, he is a Lecturer and Researcher with the City University of Hong Kong. His research interests include secure system design, network security, blockchain, and more broadly application of cryptography.

Chengjun Cai (Member, IEEE) received the Ph.D. degree in computer science from the City University of Hong Kong, in 2021. Currently, he is a Research Fellow with the City University of Hong Kong (Dongguan). His research interests included applied cryptography, data security and privacy, and blockchain.

Hongbing Cheng received the Ph.D. degree from Nanjing University of Posts and Telecommunications. Currently, he is a Professor with the College of Computer Science, Zhejiang University of Technology. His research interests include blockchain, cryptography, privacy preserving and information security, computer communications, and cloud computing security.

Ke Xu (Fellow, IEEE) received the Ph.D. degree from Tsinghua University, Beijing, China. Currently, he is a Full Professor with the Department of Computer Science, Tsinghua University. He has published more than 200 technical papers and holds 11 U.S. patents in the research areas of nextgeneration internet, blockchain systems, the Internet of Things, and network security. He serves as the Steering Committee Chair for IEEE/ACM IWQoS. He has guest-edited several special issues for IEEE and Springer journals. He is an Editor of IEEE INTERNET OF THINGS JOURNAL.

Qi Li (Senior Member, IEEE) received the Ph.D. degree from Tsinghua University. Currently, he is an Associate Professor with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include internet and cloud security, mobile security, and big data security. He is an Editorial Board Member of the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING and *DTRAP* (ACM).