DeFeed: Secure Decentralized Cross-Contract Data Feed in Web 3.0 for Connected Autonomous Vehicles

XINGCHEN SUN, Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, China

RUNHUA XU, School of Computer Science and Engineering, Beihang University, China

WEI NI, School of Electrical and Data Engineering, University of Technology Sydney, Australia

LI DUAN*, Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, China

CHAO LI*, Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, China

Smart contracts have been a topic of interest in blockchain research and are a key enabling technology for Connected Autonomous Vehicles (CAVs) in the era of Web 3.0. These contracts enable trustless interactions without the need for intermediaries, as they operate based on predefined rules encoded on the blockchain. However, smart contacts face significant challenges in cross-contract communication and information sharing, making it difficult to establish seamless connectivity and collaboration among CAVs with Web 3.0. In this paper, we propose DeFeed, a novel secure protocol that incorporates various gas-saving functions for CAVs, originated from in-depth research into the interaction among smart contracts for decentralized cross-contract data feed in Web 3.0. DeFeed allows smart contracts to obtain information from other contracts efficiently in a single click, without complicated operations. We judiciously design and complete various functions with DeFeed, including a pool function and a cache function for gas optimization, a subscribe function for facilitating data access, and an update function for the future iteration of our protocol. Tailored for CAVs with Web 3.0 use cases, DeFeed enables efficient data feed between smart contracts underpinning decentralized applications and vehicle coordination. Implemented and tested on the Ethereum official test network, DeFeed demonstrates significant improvements in contract interaction efficiency, reducing computational complexity and gas costs. Our solution represents a critical step towards seamless, decentralized communication in Web 3.0 ecosystems.

CCS Concepts: • **Security and privacy → Domain-specific security and privacy architectures**.

Additional Key Words and Phrases: Blockchain, Ethereum, Smart Contract, Data Feed, Web3, CAVs

---

*Corresponding authors: Chao Li and Li Duan

---

arXiv:2505.09928v2 [cs.CR] 19 May 2025

1 INTRODUCTION

Connected Autonomous Vehicles (CAVs) have emerged as a transformative application domain that can benefit significantly from Web 3.0 technologies like blockchain [33, 34], decentralized applications (dApps), and virtual or augmented reality. CAV networks rely on intelligent communication, coordination, and automated transactions [11]. These requirements are aligned with the vision of a more decentralized and trustless computing paradigm in Web 3.0. A crucial factor for the development of Web 3.0 ecosystems is smart contracts, which are self-executing programs on the blockchain and allow for trusted interactions without intermediaries. Smart contracts are particularly relevant for coordinating fleets of CAVs by encoding rules around negotiation, charging, access control, and other aspects of vehicle operation directly on the blockchain represented by the Ethereum. The Ethereum [7, 44, 46] has the world's largest open-source community for blockchains, and smart contracts [24, 42] are its core technology. The combination of CAVs with Web 3.0 technologies like Ethereum's smart contracts has the potential to enable secure, decentralized coordination and automation of various vehicle functions. Smart contract-based dApps running on blockchain networks could help manage charging, ridesharing, vehicle access permissions, dynamic pricing models, and other services in a trustless manner without centralized authorities. Furthermore, blockchain's append-only ledger provides a tamper-proof record of vehicle operations, transactions, and interactions, which enhances transparency and accountability.

In the context of CAV networks, the adaptivity and autonomy enabled by blockchain and smart contracts are particularly valuable. Smart contracts can dynamically adjust operational parameters based on real-time data inputs, ensuring optimal performance without human intervention. For example, smart contracts can automate the negotiation and execution of charging schedules for electric vehicles, adjusting based on current energy prices and network load. This dynamic adaptation not only improves efficiency but also reduces operational costs.

Moreover, the decentralized nature of blockchain ensures that decisions within CAV networks are made transparently and without the need for a central authority [51]. This enhances the reliability and trustworthiness if the system, as each transaction and decision is recorded on the blockchain and can be audited. Such autonomous decision-making capabilities are crucial for the scalability and robustness of CAV networks, as they enable vehicles to operate independently while adhering to globally agreed-upon rules encoded in smart contracts.

Smart contracts fundamentally alter the way that Ethereum transactions occur [23, 27, 39], due to their many unique features, such as immutability, trustless execution, and high availability. Once a smart contract is deployed, the code cannot be altered, ensuring that the original terms are preserved throughout the lifetime of the contract. The immutability prevents any party of the contract from unilaterally altering the agreement. Smart contracts execute automatically based on their programmed logic and predefined conditions. They do not require a trusted third party or intermediary, allowing anonymous parties to conduct trustless transactions. These contracts execute on-chain designed to be highly redundant and resistant to downtimes or single points of failure, ensuring high reliability. With these desirable merits, smart contracts have gained widespread trust and play a crucial role in revolutionizing several rapidly expanding industries. For instance, Decentralized Finance (DeFi) platforms, such as Uniswap [1] leverage smart contracts to provide
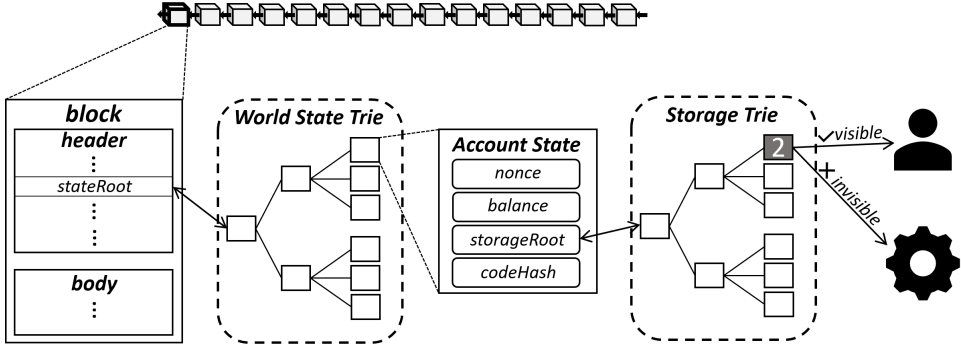
---

[1]https://uniswap.org/

Fig. 1. The Cross-Contract Data Feed (CCDF) problem. The CCDF problem is represented in this structure. It arises because a smart contract cannot directly access variables stored in another contract's Storage Trie. For example, if an integer variable with a specific value is stored in one contract's storage, another contract doesn't have direct visibility to this data.

better financial solutions. Similarly, Non-Fungible Token (NFT) [2] trading facilitated by platforms like OpenSea [2] and Rarible [3], have also witnessed significant growth due to the implementation of smart contracts.

Over the past decade, extensive research has focused on designing smart contracts to facilitate interactions between on-chain and off-chain systems. Various approaches have emerged to address data integration challenges. The data feed mechanism [38] was initially developed to provide smart contracts with access to external world information. Early systems leveraged trusted hardware and blockchain technology to authenticate and ensure the reliability of external data sources. This innovation bridged the gap between smart contracts and trusted external websites, offering a foundation for reliable data authentication. Subsequent advancements enhanced these capabilities significantly. Improved data feed services introduced advanced features, such as enhanced transparency and consistency validation, enabling better parsing of external content and stronger authenticity guarantees. Later, privacy-preserving data feed schemes emerged, incorporating advanced cryptographic techniques like zero-knowledge proofs and privacy-focused technologies to secure sensitive data while maintaining its usability in smart contract operations. These developments represent a continuous evolution of data feed mechanisms toward greater security, reliability, and functionality. Despite these advancements, existing approaches primarily focused on on-chain and off-chain interactions, overlooking a critical challenge: inter-contract data exchange. Within the Ethereum network, smart contracts remain largely isolated, unable to seamlessly share data with one another. This limitation significantly impedes potential collaborative and interconnected blockchain applications. The Cross-Contract Data Feed (CCDF) problem represents a crucial research frontier, highlighting the need for innovative solutions to enable more dynamic and integrated smart contract ecosystems. Section 3.1 provides a comprehensive analysis of this challenge, as illustrated in Fig. 1.

In this paper, we propose DeFeed, an innovative secure protocol that incorporates various gas-saving functions for solving the decentralized CCDF problem among smart contracts. The development of DeFeed is founded on our extensive research into the intricate interactions and communication mechanisms between different smart contracts deployed on the Ethereum. One of the key challenges that DeFeed addresses is the inherent limitation in Ethereum's architecture that restricts direct data access across contracts. This often requires complex operations and incurs

---

substantial gas costs for smart contracts to obtain information from one another. `DeFeed` overcomes the limitation by designing a new protocol architecture consisting of a series of interconnected smart contracts. This empowers smart contracts with the ability to seamlessly obtain information from other contracts through a simple, one-click process. As a result, `DeFeed` eliminates the need for complicated operations while maintaining low gas costs for cross-contract data feed. Furthermore, we design and implement various gas-saving and efficiency-enhancing features within the `DeFeed` protocol.

Notably, `DeFeed` incorporates a pool and cache mechanism to further optimize gas consumption. The pool function aggregates multiple requests targeting the same data source, processing them in a batch to reduce redundant computations. The cache function pre-stores and caches frequently accessed data and function call results, eliminating the need for repetitive on-chain operations. These mechanisms significantly reduce the gas expenditure required for cross-contract data feed. Additionally, `DeFeed` offers a subscribe function that enables smart contracts to subscribe to specific states from other contracts. Subscribed contracts automatically receive updates whenever the target contract's data changes, eliminating the need for repetitive querying. This subscription-based approach provides a more efficient and responsive data feed system, where contracts can stay up-to-date with the latest information without incurring the gas costs associated with constant polling. Moreover, we integrate an update function into `DeFeed`. As `DeFeed` is implemented as a series of smart contracts, it can undergo on-chain updates [4], facilitating the iterative improvement and evolution of its functionalities over time [6, 9, 12, 13, 15, 16, 20, 28, 31, 35, 47, 50].

By leveraging the pool, cache, subscribe, and update features, `DeFeed` significantly improves the cost-effectiveness and responsiveness of the cross-contract data feed process on the Ethereum blockchain, making it a more practical and scalable solution for smart contract interactions. With the `DeFeed` protocol, smart contract developers can effortlessly access cross-contract data through a simplified interface, without the burden of intricate operations. This not only enhances the development experience but also potentially unlocks new possibilities for decentralized applications by enabling efficient and secure data feed across the Ethereum ecosystem.

Contributions. The contributions of this paper are summarized as follows:

- To the best of our knowledge, `DeFeed` is the first secure and decentralized cross-contract data feed protocol designed for Ethereum-like platforms. `DeFeed` empowers smart contracts to access and exchange data from each other without compromising the core principles of decentralization, making it a pioneering protocol for CAVs with Web 3.0.
- We utilize the state-of-the-art technology to develop and implement upgradable smart contracts with various functions, such as function pool and cache for gas-saving and efficiency, and a subscribe function to provide low-cost access to information in `DeFeed`.
- We successfully implement and evaluate `DeFeed` on the Ethereum official test network. The results demonstrate that `DeFeed` is inexpensive and convenient to implement, and efficient for facilitating communication and data feed between smart contracts - a key enabler for Connected Autonomous Vehicles networks with Web 3.0.

The rest of this paper is organized as follows: we introduce the background and related work in Section 2 and provide an overview of the cross-contract data feed problem in Section 3. In Section 4, we propose some components and functions about `DeFeed`. Section 5 describes the performance evaluation of proposed `DeFeed`. Finally, we conclude the paper in Section 6.

---

[4]https://docs.openzeppelin.com/learn/upgrading-smart-contracts

2 BACKGROUND AND RELATED WORK

In this section, we explore the link between autonomous and adaptive systems and blockchain, outline the basics of various Ethereum blockchain accounts, introduce essential cryptographic tools used in our work, and discuss related work.

2.1 Autonomous and Adaptive Systems with Blockchain

Autonomous and adaptive systems [29] are designed to perform tasks without human intervention and to adjust their behavior in response to changes in their environment. These systems play a crucial role in various modern technologies, including CAVs, smart grids, and Internet of Things (IoT) networks [41]. The primary aim of these systems is to enhance efficiency, reliability, and scalability while minimizing the need for centralized control and human oversight.

Autonomous systems operate independently, making decisions based on predefined rules and real-time data inputs. They are capable of performing tasks such as navigation, coordination, and resource allocation without direct human control. Key characteristics of autonomous systems include self-governance, real-time operation, scalability, and reliability. Self-governance refers to the system's ability to make decisions and perform actions based on internal logic and external stimuli. Real-time operation ensures that the system processes and responds to data in a timely manner, facilitating accurate and prompt actions. Scalability allows the system to handle increasing amounts of work or to expand easily to accommodate growth. Reliability guarantees consistent and predictable performance under various conditions.

Adaptive systems, on the other hand, are engineered to adjust their behavior based on changes in their operating environment. This adaptivity ensures that the system remains efficient and effective even as conditions evolve. Essential features of adaptive systems include dynamic adjustments, learning and optimization, and flexibility. Dynamic adjustments enable the system to modify operational parameters and strategies in response to new data or environmental changes. Learning and optimization allow the system to improve performance over time through learning algorithms and optimization techniques. Flexibility provides the system with the ability to handle a wide range of scenarios and conditions without requiring significant reconfiguration or intervention.

Blockchain technology [21] and smart contracts are highly relevant to the development and enhancement of autonomous and adaptive systems. Blockchain offers a decentralized, secure, and transparent platform for data storage and transaction processing, which is critical for the trustless operation of these systems [45]. Smart contracts, which are self-executing programs running on the blockchain, facilitate automation and dynamic interactions among system components.

In the context of CAV networks [18], blockchain and smart contracts offer several advantages. The decentralized nature of blockchain eliminates the need for a central authority, thereby reducing single points of failure and enhancing system resilience. Blockchain's immutable ledger ensures that all transactions and data are secure and verifiable, fostering trust among participants. Smart contracts automate various processes, such as vehicle coordination, charging, and access control, based on predefined rules and real-time data inputs. Furthermore, smart contracts can dynamically adjust their operations in response to changes in the environment, such as fluctuating energy prices or varying traffic conditions, thereby enhancing the system's adaptivity.

2.2 Preliminaries for Ethereum

External Owned Accounts: External Owned Accounts (EOAs) are Ethereum accounts that are controlled by private keys outside the Ethereum network. This is the most common type of account on the Ethereum blockchain, and individuals commonly use it to interact with the Ethereum network, store Ether and other ERC-20 tokens, and send or receive transactions.

EOAs are identified by their Ethereum address, a 20-byte hexadecimal string. When an EOA initiates a transaction on the Ethereum network, it must sign the transaction with its private key and broadcast it to the network. Once the transaction is confirmed, the Ethereum network updates the state of the account accordingly. EOAs differ from Contract Accounts (CAs), which are accounts associated with smart contracts deployed on the Ethereum network. CAs are controlled by the smart contract's code, rather than by a private key, and are used to store and manage assets or execute complex transactions.

In summary, EOAs are a fundamental and essential component of the Ethereum network as they allow for secure and decentralized interaction between individuals and the network.

Transactions: Ethereum supports two main types of transactions: regular value transfers and contract interactions. Regular transactions involve the simple transfer of Ether from one account to another. On the other hand, contract interactions require the invocation of functions or execution of code within smart contracts deployed on the Ethereum blockchain [25].

Each transaction in Ethereum comes with a unique identifier called a nonce that prevents double-spending attacks and ensures that transactions from a specific account are executed in the correct order [26]. The nonce increments with each transaction sent from an account. When it comes to executing transactions in Ethereum, the gas limit is a crucial factor to consider. It prevents the maximum amount of gas that a sender is willing to spend on a transaction, which determines the computational resources, such as CPU time and memory, allocated to execute the transaction. If the gas limit is set too low, the transaction may run out of gas and fail. Hence, setting an appropriate gas limit is crucial to ensure that transactions execute successfully.

Another essential factor to consider is the gas price. It refers to the amount of ether that the sender is willing to pay per unit of gas used in the transaction. Miners prioritize transactions with higher gas prices, as they provide greater incentives for including them in blocks. Moreover, gas prices are not static and can fluctuate depending on the level of network traffic and the prevailing demand in the market.

Transaction fees in Ethereum are calculated as

$$TotalFee = (baseFee + priorityFee) * gasPrice \tag{1}$$

The total fee is determined by multiplying the gas price by the actual gas consumed during transaction execution. Gas consumption depends on the complexity of the transaction or smart contract operation. When a transaction is initiated, it is broadcast to the Ethereum network and propagated to nodes. Miners then include pending transactions in blocks by solving complex mathematical puzzles through the mining process. The blockchain consensus mechanism guarantees that a valid transaction takes at most θ time to be recorded on the ledger [34, 48, 49]. The time delay θ is also known as the blockchain delay. Once a block is mined and added to the blockchain (e.g., the ledger), the transaction is considered confirmed. While transactions are considered final once they are confirmed and added to the blockchain, it's essential to note that Ethereum employs a probabilistic finality mechanism. This means that as more blocks are added to the blockchain after a transaction's inclusion, the probability of its reversal decreases significantly. It can only be changed through a network-level attack that costs billions of dollars.

Ethereum Virtual Machine: The Ethereum Virtual Machine (EVM) is a key component of the Ethereum blockchain that enables the execution of smart contracts and decentralized applications (DApps). It serves as a decentralized runtime environment for executing code written in Ethereum's native programming language, Solidity [5], as well as other compatible languages like Vyper [6].

---

[5]https://github.com/ethereum/solidity
[6]https://github.com/vyperlang/vyper

The EVM is a decentralized execution environment that runs on thousands of nodes in the Ethereum network. Each node keeps a copy of the Ethereum blockchain and executes smart contracts and transactions independently. Smart contracts are deployed on the Ethereum blockchain and are converted into bytecode, which is a low-level representation of the contract's instructions. The EVM then executes this bytecode sequentially, interpreting each operation code and performing the corresponding action. These operation code sequences are generated from the high-level smart contract code written in Solidity or other compatible languages. The EVM guarantees deterministic execution of smart contracts, which means that when given the same input and initial state, the outcome of contract execution is always the same. This property is crucial for achieving consensus among nodes on the network, as all nodes must arrive at the same result when executing transactions and smart contracts.

GAS: On the Ethereum network, users pay Ethereum gas [10] to process transactions or use smart contracts. Ethereum gas is measured in gwei, which is short for gigawei, with one gwei being equivalent to one billionth of an ETH. However, Ethereum gas fees can only be paid using Ether, which is the platform's native token.

It's crucial to note that a malicious user posing as a legitimate one can execute resource-exhausted code in a function, leading to a Denial of Service (DoS) attack that could harm the entire network. This is because every single node in the network has to execute every function call to achieve a consensus on the state of balances and contracts.

To safeguard against such attacks, each fundamental atomic operation has been assigned a specific amount of gas. Each Ethereum transaction has a specified gas limit, which represents the maximum amount of gas the sender is willing to pay for the transaction. Gas price is the amount of Ether paid per unit of gas and is set by the sender. The total transaction fee is the product of the gas limit and gas price. Hence, whenever a function is called, the initiator of the call must pay a transaction fee that covers the total gas consumption of the on-chain computations involved in the call. As an incentive, the fee above is paid to the miner, not to each individual node.

While gas has been effective in preventing DoS attacks, it has also led to a significant increase in transaction fees for users of blockchain [8]. According to the statistics of the Ethereum blockchain, the largest programmable blockchain in terms of market capitalization [7], users had to pay an average of 5,369,200 ETH (approximately 10 billion USD) per day in gas costs in the last year. The official documentation of the Ethereum Foundation has acknowledged the issue of high gas fees [8].

Due to the high fees, several smart contracts have limited functionality and rely on simple programs where each function ends in a constant number of steps. Additionally, there are several protocols, including layer-two protocols, that sacrifice decentralization or trustlessness in order to reduce gas costs.

2.3 Cryptographic tools

Keccak-256 hash function: The Keccak-256 hash function belongs to the Keccak family of cryptographic hash functions. It gained widespread recognition for being chosen as the algorithm for Secure Hash Algorithm 3 (SHA-3) by the National Institute of Standards and Technology (NIST) in 2012. However, it's worth noting that Ethereum's use of Keccak-256 precedes the final standardization of SHA-3, and as a result, it differs slightly from the SHA-3-256 variant standardized by NIST.

In Ethereum, Keccak-256 is used extensively for various purposes. Creating Addresses, Ethereum addresses are derived by taking the Keccak-256 hash of the public key derived from the private key

---

[7]https://coinmarketcap.com/
[8]https://ethereum.org/en/developers/docs/gas/

and then taking the last 40 characters (20 bytes) of that hash. Transaction Hashing, transactions are identified by their Keccak-256 hash. Smart Contracts, the bytecode of Ethereum smart contracts is also hashed using Keccak-256. This hashing facilitates the EVM operation and integrity verification. Merkle Patricia Trees (MPT), Ethereum uses a modified version of a Merkle Patricia Tree for its blockchain state, where Keccak-256 is used to hash the nodes and leaves of the tree, ensuring integrity and enabling efficient and secure data verification.

ECDSA: Many mainstream blockchain systems, such as Ethereum and Bitcoin, adopt the Elliptic Curve Digital Signature Algorithm(i.e., ECDSA). ECDSA is a public key cryptography algorithm used to verify the validity of digital signatures. It is based on elliptic curve cryptography and uses private and public keys to generate digital signatures and verify their validity. In Ethereum, ECDSA is used to verify whether the sender of a transaction has the authority to send it. Specifically, when a user sends a transaction, they must sign it with their private key and then broadcast the signature and transaction together on the network. Other nodes can use the sender's public key to verify the validity of the signature, ensuring that the transaction was sent by its rightful owner. In Ethereum, ECDSA is also used to generate Ethereum addresses. An Ethereum address is generated from a public key using hash functions, so ECDSA algorithms are needed for generating public keys. Typically, an Ethereum address consists of 40 hexadecimal characters in string format, such as 0x7cB57B5A97eAbe94205C07890BE4c1aD31E4XXXX.

2.4 Upgradeable Smart Contracts

Immutability or tamper-proof is a major property of blockchain systems, like Ethereum. Smart contracts that are deployed on such blockchain systems also possess this property. Smart contracts play an important role in the operation of blockchain systems with this advantage.

But they also have certain flaws and limitations. The primary flaw is that they can not be upgraded as general software does. Once deployed on the blockchain systems, these contracts can not be altered or upgraded. This means that any bugs, vulnerabilities, or shortcomings discovered after deployment cannot be easily rectified without deploying an entirely new contract. Bugs or errors discovered in the smart contract code after deployment cannot be easily fixed in non-upgradeable contracts. Users and developers may need to resort to deploying an entirely new contract, which can be a cumbersome and costly process. If there is a need to migrate to a new contract (e.g., due to security concerns or major updates), users may face challenges in moving their assets and data from the old contract to the new one. This process can be complex and may result in fragmentation of the user base. In the event of a security vulnerability, non-upgradeable contracts may pose significant risks. Without the ability to patch or upgrade the contract, users' funds and assets may be at risk, and exploits could lead to irreversible consequences. How to design upgradeable smart contracts is a key issue of concern in both academic and industry circles.

The proxy pattern [3, 5] is a great design that smart contracts can use to be upgradeable. It gives developers some kind of leeway to modify contract logic post-deployment. In a proxy-contract architecture, there are two main parts:

1) The proxy contract,
2) The execution or logic contract.

The proxy contract is deployed at the beginning and contains the contract storage and balance. The execution contract stores the contract logic that is used to execute functions. The proxy contract stores the address of the execution contract. When users send requests, the message goes through the proxy contract. The proxy contract then routes the message to the execution contract, which performs the computation. Afterward, the proxy contract receives the result of the computation from the execution contract and returns it to the user. It is important to note that the proxy contract

itself cannot be modified. However, additional execution contracts with updated contract logic can be created, and message calls can be rerouted to the new contract. With the proxy pattern, you are not changing the smart contract itself. Instead, you are deploying a new logic contract and asking the proxy contract to reference it instead of the old contract. This new logic contract can have different functionality than the previous one or fix an old bug.

2.5 Related Work

In recent decades, blockchain technology has gained widespread adoption across various industries. One of its key features is the implementation of smart contracts, which enable the execution of business logic in a decentralized structure, resulting in trusted and universally accepted outcomes among participating nodes. However, smart contracts face a fundamental limitation: they cannot independently access external data sources. To address this, they rely on oracles - off-chain external data sources - to collect and provide data feeds and inputs [52], [1].

Extensive research has been conducted on blockchain oracles and data feed mechanisms. Zhang et al. [52] introduced TownCrier, a platform leveraging hardware-based Trusted Execution Environments (TEEs), such as Intel's SGX enclaves, to collect data from HTTP-enabled sources and provide authenticity proofs securely. This approach significantly enhanced the trustworthiness of external data inputs to smart contracts. Building on this foundation, Juan et al. [17] proposed the Practical Data Feed Service (PDFS) system, which establishes a robust connection between smart contracts and external data sources. PDFS addresses scalability and efficiency concerns in oracle systems, offering a more practical solution for real-world applications.

Recognizing the importance of privacy in blockchain data feeds, Wang et al. [43] developed an innovative on-and-off-chain data feed privacy-preserving scheme. This approach incorporates zero-knowledge proofs [14] and Hawk technology [22], enabling smart contracts to interact with external data while maintaining strong privacy guarantees. The reliability and trustworthiness of oracle systems have been a focal point of research. Lo et al. [30] conducted a comprehensive analysis of trust-enabling features across various blockchain oracle approaches, techniques, and platforms. Their work provides valuable insights into the strengths and weaknesses of different oracle solutions, guiding future developments in the field.

Recent developments in the field include the exploration of multi-chain oracle solutions to enhance interoperability between different blockchain networks. Additionally, researchers are investigating the use of artificial intelligence and machine learning techniques to improve the accuracy and reliability of data feeds.

Blockchain technology has been widely recognized for its potential to enhance the functionality of autonomous systems by providing a decentralized and secure platform for communication and decision-making. For instance, Menha et al. [32] explored the use of blockchain in autonomous vehicles, highlighting how blockchain can ensure secure data sharing and trust among vehicles. Ren et al. [37] addresses the security of IoT devices' sampled data in agriculture, proposing a double-blockchain solution using IPFS storage. Rathee et al. [36] employ blockchain technology to solve the various security challenges in connected vehicles.

Smart contracts, which are self-executing programs on the blockchain, play a crucial role in enabling adaptivity in various systems. Baza et al. [4] employed the blockchain and smart contract that proposes a distributed firmware update scheme for autonomous vehicles. Jain et al. [19] discussed how smart contracts could automate processes in supply chain management, adapting to new data inputs and conditions without human intervention. In the context of Internet of Vehicles (IoV), Sigh et al. [40] introduces a blockchain-based decentralized trust management scheme for IoV using smart contracts to address security and privacy threats.

To sum up, current techniques and tools for data feed in blockchains focus on enabling data interaction between on-chain and off-chain sources. Our work in this paper focuses on the data feed between smart contracts on-chain and tackles the decentralized CCDF problem. To the best of our knowledge, DeFeed is the first secure and decentralized data feed protocol designed for Ethereum-like platforms and CAVs with Web 3.0.

3 THE CROSS-CONTRACT DATA FEED PROBLEM

In this section, we provide an overview of the CCDF problem. We then introduce two existing architectures for solving this problem and propose a new solution employed by DeFeed.

3.1 Overview

The CCDF problem is essentially induced by the lack of visibility of a smart contract to variables stored in another smart contract. In Fig. 1, we illustrate where an integer variable with the value is stored in Ethereum. When a new block is created, each miner must synchronize with the most recent block and update the local database storage (i.e., LevelDB in Ethereum) accordingly. A block in Ethereum consists of a header and a body, with the header containing several elements, including the roots of three MPT trees, namely the root *stateRoot* for the World State Trie tree and two other roots for the Receipts Trie tree and the Transactions Trie tree, respectively. In the World State Trie tree, each leaf node represents a snapshot of the current state of an EOA or a CA. The state of an account consists of four components:

$$state \equiv \langle nonce, balance, storageRoot, codeHash \rangle \tag{2}$$

where *nonce* indicates the number of transactions sent from the account, *balance* denotes the current account balance, and *codeHash* represents the hash value of the EVM bytecode of the smart contract in case the account is a CA. Here, *storageRoot* refers to the root of the Account Storage Trie tree for a CA, where the current values of all smart contract variables are stored.

The state of variables in smart contracts can be considered visible to miners because encoded values of these variables are physically stored in LevelDB, and miners can either use the verified contract code provided by platforms, such as Etherscan, or reverse engineering tools to decode and read the values. In addition, public variables have built-in getters, so EOAs can always access them. In contrast, private/internal variables might not be directly accessible to EOAs, although they could be made accessible to EOAs through explicit getters.

Miners and EOAs are essentially two roles people play, so they can read values of variables from smart contracts using various off-chain techniques as long as the blockchain data is available. Unlike miners and EOAs, smart contracts, or CAs, are like the residents living in the blockchain world; hence, they have to obey the rules made by the developers of the blockchain system. Unfortunately, in Ethereum, one critical rule is that a smart contract is unaware of the state of variables in another smart contract. In other words, by default, the storage of a smart contract is invisible to another smart contract, resulting in the CCDF problem.

3.2 Architectures for CCDF

In Ethereum, the traditional data interaction process can be implemented through different architectural approaches, as illustrated in Fig. 2.(a) and Fig. 2.(b). Here, C_c, C_d, C_e and C_f represent smart contracts that own specific data, while C_a and C_b represent smart contracts seeking to request data.

The direct data feed architecture, shown in Fig. 2.(a), establishes a direct connection between contracts like C_a and C_d. This architecture enables immediate data exchange between contracts while maintaining low gas costs. However, this approach raises significant privacy concerns due to its lack of access control mechanisms.
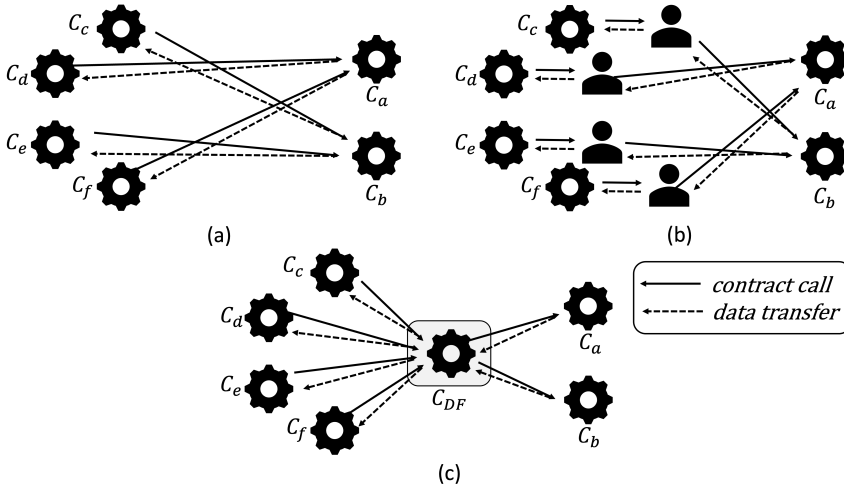
Fig. 2. The traditional data feed architecture (a), the third-party data feed architecture (b), and the cross-contract data feed architecture (c). In this figure, each gear represents a smart contract, each persona represents a third party. Solid arrows indicate "contract call". while dashed arrows represent "data transfer".

Fig. 2.(b) presents an alternative architecture that introduces a third party to mediate data feed between smart contracts. While this approach enables monitoring of data legality, it introduces additional costs and creates a single point of trust. Furthermore, the reliance on a centralized third party may lead to availability issues, potentially causing delays in data delivery.

To address these limitations, we propose a novel architecture illustrated in Fig. 2(c). This design replaces the third party with a dedicated smart contract (C_{DF}) that serves as a secure data feed hub. Data-providing contracts can register with C_{DF} using contract names or specific nicknames, creating a binding between these identifiers and their addresses. When contract C_a needs data from C_d, it sends a request to C_{DF} containing the registered name of the target contact. C_{DF} then resolves the address and facilitates the data exchange, maintaining isolation between data owners and requestors while ensuring secure and efficient data transfer.

3.3 The chainedCALL pattern

To implement the proposed architecture, we employ a fundamental design pattern based on the chained opcode CALL. This pattern, which we term *chainedCALL*, enables controlled data access between smart contracts through explicit function provisioning.

Fig. 3 demonstrates the elementary implementation of this pattern through three dedicated functions: *request*, *response*, and *receive*. The process begins when an EOA initiates a function call to *request* in the data requestor contract. This triggers a chain of CALL operations, first invoking *response(·)* in the data owner contract, which then calls *receive(·)* in the requestor contract to complete the data transfer.

The chainedCALL pattern serves as the basic architectural building block for CCDF, represented by turn-around lines with arrows between contracts as shown in Fig. 3. While this example presents a simplified implementation, real-world applications require additional logic for error handling, data validation, and security measures to ensure reliable operation of CAVs and their interactions.
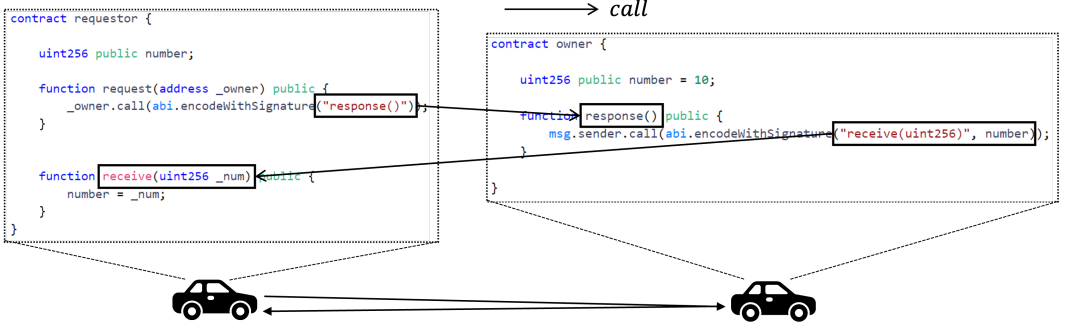
Fig. 3. Regular data feed process. Solid arrows represent contract function calls. The request function in the requester contract calls the response function in the owner contract, which then calls the receive function in the requester contract.

By combining our proposed architecture with the chainedCALL pattern, DeFeed achieves a balance between security, efficiency, and practical implementation requirements for cross-contract data feed operations.

In the rest of this paper, we consider the above pattern of the chained opcode CALL as the basic architecture building block for CCDF. For simplicity, we name the pattern *chainedCALL* and denote it by turn-around lines with arrows between two smart contracts, as shown in Fig. 3.

4 THE PROPOSED METHOD: DEFEED

In this section, we first present the basic protocol called Data Feed, which enables data feed across different smart contracts. Next, we present more complex protocols involving the function *Pool* and *Cache* based on the basic protocol. Additionally, we explain how the protocol upgrades core components using the *Update* function and how smart contracts can receive notifications when the subscribed smart contract updates its status with the function *Subscribe*.

4.1 Data feed

To establish a secure communication protocol among smart contracts, we first devise and implement a fundamental protocol called the Data Feed. To improve isolated management and facilitate future protocol iterations, we modified the architecture from Fig. 2(c) to the overall framework depicted in Fig. 4 for DeFeed. We split one smart contract C_{DF} into two separate smart contracts and create a decentralized committee in order to avoid the centralization of our protocol. One of the smart contracts is responsible for management, while the other is responsible for implementing the basic functions of the protocol. We sketch the protocol in Fig. 5 and present a formal description of data feed interactions.

The fundamental protocol comprises five primary components: a Data Owner Contract (C_o), a Requestor Contract (C_r), a Data Feed Center Contract (C_{DFC}), a Data Feed Management Contract (C_{DFM}), and a Committee. The Committee is a collective of individuals responsible for managing C_{DFM}. The contract C_o represents a CAV that provides data, while the contract C_r represents a CAV that requests data. The smart contracts C_{DFC} and C_{DFM} are the core part of DeFeed, facilitating data feed between parties. We formally define the workflow system as a triple:
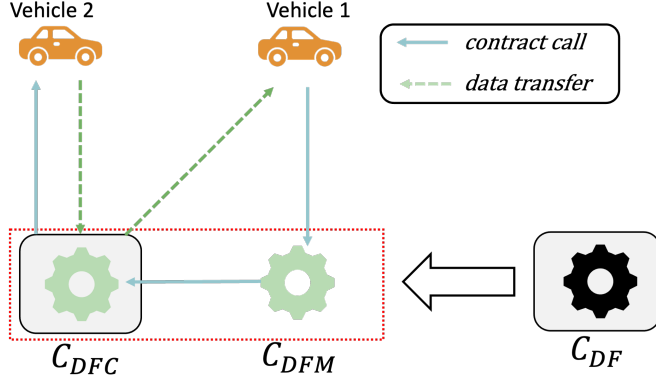
$$W = (C, I, M) \tag{3}$$

Fig. 4. Overall framework of DeFeed. The smart contract C_{DF} is divided into two separate contracts. Solid blue arrows represent "contract call", while the dashed green arrows indicate "data transfer". Vehicle 1 requests data from Vehicle 2 using the complex contracts outlined in the red dashed box. A red box indicates that its interior is isolated from the exterior.
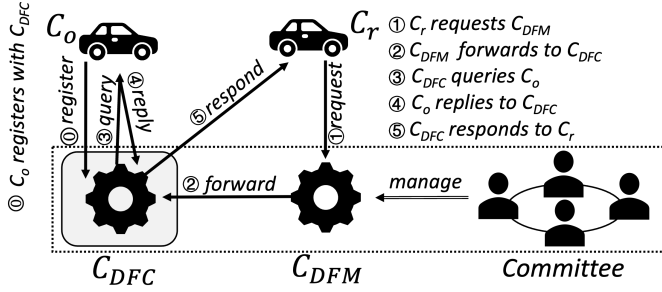

Fig. 5. Detailed workflow of the regular data feed process. A contract C_{DFC} receives registration data from a vehicle C_o (Step 0). Another vehicle C_r can make requests to a contract C_{DFM} contract for accessing the registered data (Step 1). C_{DFM} will forward the request to C_{DFC} (Step 2). C_{DFC} queries C_o for the data (Step 3&4) and responds it to C_r (Step 5). The committee manages the overall process.

where C is the set of components $C = \{C_o, C_r, C_{DFC}, C_{DFM}\}$, I represents the interactions between these components, and MM denotes the management rules applied by the Committee. Each interaction in the set I signifies an action, such as a request initiation by C_r, a request forward by C_{DFM}, or a response by C_{DFC}. The Committee's management role is crucial as it shapes the behavior and decision-making process within C_{DFM}, thereby influencing the overall efficiency of the protocol. Next, we will provide a detailed description of the protocol.

Register. We define γ as an attribute tuple of smart contracts consisting of three elements (x, y, z). Among the attributes, $\gamma.x \in \{0, 1\}$ is a flag to determine whether the contract is registered, $\gamma.y$ denotes the contract identifier of a contract and $\gamma.z$ means the address of the contract. To simplify the presentation, we assume that there is one requestor contract C_r and one data owner contract C_o. To utilize our protocol, C_o is required to submit a registration request to C_{DFC}. Once the request is received, C_{DFC} will generate a list associated with C_o, which can be indexed by other contracts, like C_r.

Initially, a data owner C_o must register with C_{DFC}. The registration is send as :

$$\alpha_{reg} = (I_{reg}, C_o, A_o) \tag{4}$$

where α_{req} represents the registration request messgage, $I_{reg} \in \{0, 1\}$ represents the registration interaction identifier. C_o is the contract identifier of C_o in the protocol, A_o is the address of C_o. Upon receiving this request, C_{DFC} updates the tuple for C_o:

$$\gamma_o = (1, C_o, K(A_o)) \tag{5}$$

where γ_o is the attribute tuple for C_o and $K(\cdot)$ is a keccak256 cryptographic hash function. Only registered data owners can be discovered by requestor contracts.

Request. First, the requestor C_r creates a request with the operation $I.req$, the name of C_o and its own address. Then, C_r sends this request $(I_{req}, C_o, K(A_r))$ to C_{DFM}, waiting for the data feed.

Forward. For better security and iteration of DeFeed, our protocol has two pivotal smart contracts - a proxy contract (C_{DFM}) and a logic contract (C_{DFC}) that help to separate the data owner and requestor contracts. The C_{DFM} contract is purely a proxy contract and does not have any logical functions. When it receives a request from C_r, it will check that the requestor contract C_r has the necessary permissions to send the request. Once this check is passed, it will forward this request as $(I_{for}, C_o, K(A_r))$ to the C_{DFC} contract.

Query. Upon receiving $(I_{for}, C_o, K(A_r))$ from C_{DFM}, C_{DFC} will search in γ based on the passed username C_o. If $\gamma.x$ is 0, it means that this name Co is not registered yet and C_{DFC} will send $(I_{resp}, K(A_r)), \beta)$ to C_r, where β is defined as a string and contains "The name doesn't exist.". Otherwise, C_{DFC} will send (I_{query}, C_o) to C_o using the address of C_o that is set in γ before.

Reply. Upon receiving the (I_{query}, C_o), C_o will verify the sender's address. If the sender's address matches C_{DFC}, C_o will execute the $reply(\cdot)$ operation. In this operation, C_o will retrieve the requested data and package it into the (I_{reply}, κ) response, where κ is the encrypted data. C_o will then send this response back to the C_{DFC} contract. However, if the sender's address does not match C_{DFC}, the *accessOnly(·)* function in the code logic of C_o will deny the I_{query}, preventing unauthorized access to the data.

Respond. After processing the (I_{reply}, κ) response, the C_{DFC} contract will perform several validation checks. It will verify that the response comes from C_o. If the response is valid, C_{DFC} will package the data into a final (I_{resp}, κ) message and send it back to the original requestor contract C_r. Then C_r can retrieve and use the data that was requested from C_o. The C_{DFC} contract also records information about the request and response in its internal logs or state for auditing and monitoring purposes.

Next, we present the protocol with different functionalities for data feed and discuss how the four key components in the set C work in various functionalities. We start by introducing the protocol with pool shown in Fig. 6(a), where C_{DFM} can pool multiple requests and only send one request through function *forward(·)*. We then introduce the protocol with cache in Fig. 6(b), where C_{DFM} can cache any successful results from the previous contracts' request by the function *respond(·)* in C_{DFC}. When the result of another user request is found in C_{DFM}'s cache, the data can be obtained directly from the cache. Finally, we will respectively introduce the method that we use to realize the iteration of our protocol in the future and how users can subscribe to a target contract in the protocol.

Pool   To enhance the efficiency of data feed between different CAVs while reducing gas costs, we design the pool function that pools the requests with the same objective. As outlined in Section 4.1, the data feed with pool function also has the same four key components in the set C, but the functionality is kind of different in detail. In this scenario, we define a new set of the
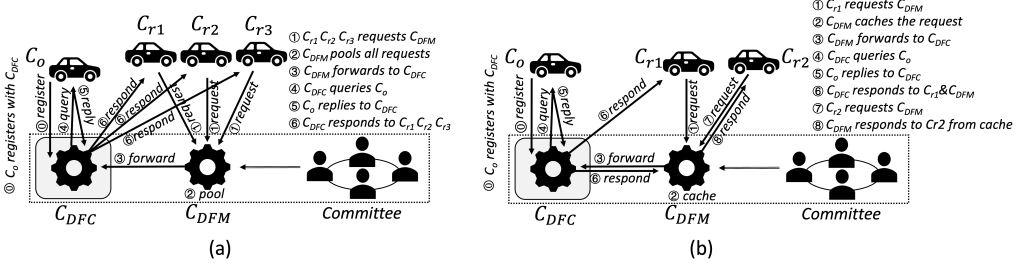
Fig. 6. The protocol for Data Feed. (a) Pool process, and (b) Cache process

requestor contracts, $CR = \{C_{r1}, C_{r2}, , C_{rn}\}$ and redefine the attribute tuple γ to include an additional element:$(\gamma.x, \gamma.y, \gamma.z, \gamma.t)$, where $\gamma.x \in \{0, 1\}$ indicates registration status, $\gamma.y$ is the registered username, $\gamma.z$ is the hashed address of the contract, and $\gamma.t$ specifies a transaction delay in seconds. Initially, we set $\gamma.z$ to 12 seconds as the transaction delay, which is the time of one block submitted to the blockchain. For clarity, we narrow down the number of request contracts to three, e.g., $CR' = \{C_{r1}, C_{r2}, C_{r3}\}$. The pool function operates as follows.

Register. In this phase, C_o submits a registration request to C_{DFC}. Upon receiving the request, C_{DFC} updates γ for C_o as follows:

$$\gamma_o = (1, C_o, K(A_o), -1) \tag{6}$$

where $\gamma.t = -1$ indicates infinite waiting time, ensuring flexibility for initial registration.

Transmission. The transmission phase begins with a request from the first requestor, C_{r1}:

$$\alpha_{req1} = (I_{req}, C_o, K(A_{r1})) \tag{7}$$

Upon receiving this request, C_{DFM} updates $\gamma.t$ for C_o to 3 blocks, creating a time window of three blocks for other requests targeting C_o. During this time, other requestors, C_{r2} and C_{r3}, send their requests:

$$\alpha_{req2} = (I_{req}, C_o, K(A_{r2})) \tag{8}$$

$$\alpha_{req3} = (I_{req}, C_o, K(A_{r3})) \tag{9}$$

If these requests arrive within the time window, C_{DFM} aggregates them into a single request:

$$\alpha_{for} = (I_{for}, C_o, K(A_{r1}, A_{r2}, A_{r3})) \tag{10}$$

This aggregated request is forwarded to C_{DFC}.

Upon receiving the forwarded request, C_{DFC} searches for the name Co in its registry. If $\gamma.reg = 0$, C_{DFC} sends an error message back to C_r:

$$\alpha_{resp} = (I_{resp}, K(A_r), \beta) \tag{11}$$

Here, β is a predefined string containing "The name doesn't exist." In the case that γ_x is not 0, this indicates that the name Co is already registered, C_{DFC} proceeds to send the message (I_{query}, C_o) to C_o. The address of C_o used for sending this message is retrieved from γ.

Note that this transmission phase contains the forward, query, and reply phases.

Respond. Finally, C_{DFC} processes the response and sends it back to each requestor:

$$\alpha_{resp} = (I_{resp}, \kappa, K(A_r)) \tag{12}$$

The hashed address allows C_{DFC} to identify and deliver the response to the corresponding requestors C_{r1}, C_{r2} and C_{r3}.

Compared with the ordinary data feed mentioned, this way of processing requests can greatly shorten processing time and save gas cost. By aggregating requests, the pool function enhances the efficiency and scalability of the data feed protocol while reducing the overall gas costs for participating contracts.

Cache   To further optimize gas consumption and improve the efficiency and scalability of the data feed protocol, we introduce a cache function based on the regular data feed. This function leverages caching to reduce redundant interactions with the Data Owner Contract (C_o) by storing responses for subsequent reuse.

We redefine the attribute tuple γ to include additional attributes for caching ($\gamma.x, \gamma.y, \gamma.z, \gamma.d$), where $\gamma.x \in \{0, 1\}$ indicates whether a cache exists for a given contract, $\gamma.y$ is the registered username of the contract, $\gamma.z$ is the hashed address of the contract and $\gamma.d$ stores the cached data. The cache function uses the *createCache(·)* operation to generate caches indexed by the username of the Data Owner Contract. For clarity, we limit the number of requestors to two: $CR' = \{C_{r1}, C_{r2}\}$. Below, we describe the cache function process from the perspectives of C_{r1} and C_{r2}.

Perspective CR1. When requestor C_{r1} initiates a request:

$$\alpha_{req} = (I_{req}, C_o, K(A_{r1})) \tag{13}$$

the request follows the regular phases of the data feed protocol: **request**, **forward**, **query**, **reply**, and **respond**. During the respond phase, C_{DFC} sends a response to both C_{r1} and C_{DFM}:

$$\alpha_{resp} = (I_{resp}, \kappa, K(A_{r1})) \tag{14}$$

Upon receiving the response, C_{DFM} executes the *create Cache(·)* function to create a cache entry for C_o:

$$\gamma_o = (1, C_o, K(A_o), \kappa) \tag{15}$$

where $\gamma.x = 1$ indicates the cache is now active and $\gamma.d = \kappa$ stores the response data.

This caching ensures that future requests for C_o can bypass redundant protocol phases.

Perspective CR2. When requestor C_{r2} initiates a similar request:

$$\alpha_{req} = (I_{req}, C_o, K(A_{r2})) \tag{16}$$

C_{DFM} checks whether a cache for C_o exists by evaluating $\gamma.x$. If $\gamma.x = 1$, C_{DFM} directly sends a cached response to C_{r2}:

$$\alpha_{resp} = (I_{resp}, \kappa, K(A_{r2})) \tag{17}$$

This skips the **forward**, **query**, **reply**, and part of the **respond** phases, significantly reducing gas costs and latency. If $\gamma.x = 0$, the request follows all phases of the data feed protocol. Once the process completes, a cache is created for future use.

Update   As blockchain technology evolves, smart contracts are becoming increasingly sophisticated and capable of executing more complex transactions. However, like any software, smart contracts require periodic updates to enhance functionality, improve efficiency, and address security vulnerabilities. In DeFeed, we have implemented a robust update mechanism to ensure the protocol remains secure, efficient, and adaptable to changing requirements.

The update function in DeFeed serves multiple critical purposes. Primarily, it allows for security enhancements by enabling the patching of potential vulnerabilities identified through ongoing security audits. This proactive approach is crucial in maintaining the integrity of financial transactions handled by DeFeed. Furthermore, the update mechanism facilitates functionality improvements, allowing for the addition of new features and enhancement of existing ones. This not only improves user experience but also attracts more smart contracts to our protocol.
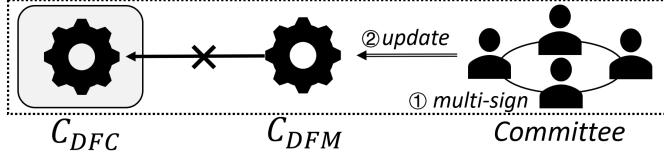
Fig. 7. Update process

Adaptability is another key benefit of the update function. As the blockchain ecosystem evolves, this mechanism ensures DeFeed can adapt to new standards, regulations, or technological advancements. Additionally, updates can include optimizations that improve the protocol's efficiency, reducing gas costs and enhancing overall performance.

The update process in DeFeed is designed to be secure, efficient, and transparent. It begins with a multi-signature approval from the committee, ensuring that updates are consensual and aligned with the protocol's governance structure. Once approved, the Contract Data Feed Manager (C_{DFM}) executes the update function. This involves deactivating the previous Contract Data Feed Core (C_{DFC}) and replacing its address with the new one using the function $update(addr(newCDFC))$.

Following the update, extensive testing is conducted to ensure the new C_{DFC} functions correctly and maintains backward compatibility where necessary. Users and stakeholders are then notified of the update, including any new features or changes in functionality.

The update mechanism in DeFeed is designed to have minimal disruption, ensuring a smooth transition to the new version with little to no downtime. Data integrity is maintained throughout the process, with all historical data and ongoing transactions preserved. This approach allows DeFeed to scale effectively, accommodating growing user bases and evolving blockchain ecosystems.

Fig. 7 illustrates the detailed update process, highlighting the role of the committee, C_{DFM}, and the transition from the old to the new C_{DFC}. By implementing this comprehensive update mechanism, DeFeed ensures long-term viability, security, and adaptability in the dynamic landscape of blockchain technology. This approach not only addresses current needs but also positions the protocol to evolve alongside future technological advancements in the blockchain space.

Subscribe We now present a new function based on the regular data feed named subscribe. The subscribe process is shown in Fig. 8 and can be represented by the following equation:

$$subscribe(C_r, C_o, S) \rightarrow N(\Delta S) \tag{18}$$

Where C_r is the receiver contract (the one subscribing), C_o is the observed contract (the one being subscribed to), S represents the state of C_o, ΔS represents any change in the status of C_o, and $N(\Delta S)$ is the notification containing details of the change.

The state S of contract C_o can be represented as a tuple:

$$S = (v_1, v_2, \ldots, v_n) \tag{19}$$

where v_i represents the i-th variable or property of the contract state. When a change occurs, we can represent the new state S' as:

$$S' = (v'_1, v'_2, \ldots, v'_n) \tag{20}$$

The change in state, ΔS, can then be calculated as:

$$\Delta S = S' - S = (v'_1 - v_1, v'_2 - v_2, \ldots, v'_n - v_n) \tag{21}$$

The notification function $N(\Delta S)$ can be defined as:

$$N(\Delta S) = \{(i, \Delta v_i) \mid \Delta v_i \neq 0, i \in [1, n]\} \tag{22}$$

This function returns a set of pairs, where each pair consists of the index of the changed variable and its change value, for all non-zero changes. In social media, we receive notifications when users update their status. Our *subscribe(·)* function operates on a similar principle within the blockchain ecosystem. Under the supervision of the committee, other smart contracts, like C_r, can subscribe to the status of contract C_o through the *subscribe(·)* function. The subscription process can be formalized as:

$$\text{Subscriptions}(C_o) = \{C_r \mid \text{subscribe}(C_r, C_o, S) \text{ has been called}\} \tag{23}$$

This set represents all contracts that have subscribed to C_o. Once there is any change (ΔS) in the status of contract C_o, then for each $C_r \in \text{Subscriptions}(C_o)$, C_r will immediately receive a notification $N(\Delta S)$, which contains the details of the change.

In social media, we receive notifications when users update their status. Our *subscribe(·)* function operates on a similar principle within the blockchain ecosystem. Under the supervision of the committee, other smart contracts, like C_r, can subscribe to the status of contract C_o through the *subscribe(·)* function.

Once there is any change (ΔS) in the status of contract C_o, then C_r will immediately receive a notification N(ΔS), which contains the details of the change. This makes it convenient for other smart contracts to know the changes in the status of the target contract anytime and anywhere, without having to check the contract status themselves.

This makes it convenient for other smart contracts to know the changes in the status of the target contract anytime and anywhere, without having to check the contract status themselves. This subscribe mechanism creates a direct link between contracts, enabling efficient information flow and improving inter-contract communication. It allows smart contracts to stay informed about relevant changes in other contracts' states in real-time, enhancing overall system efficiency and responsiveness.

4.2   Security analysis

In this section, we present the security analysis for `DeFeed`. Our data feed protocol consists of a series of distinct smart contracts, enabling smart contracts to acquire the data they want from others utilizing the protocol at a minimal cost.

Lemma 1. *The data feed management contract C_{DFM} exclusively accesses the data feed center contract C_{DFC}, which in turn solely accesses the data owner contract C_o.*

Proof. In Fig. 5, the order of access begins with the requestor smart contract C_r, which is then passed on to the data feed management contract C_{DFM}. Next, the request sent by the requestor contract is forwarded to the data feed center contract C_{DFC}. Finally, the access request goes to the data owner contract C_o and returns to its starting point, completing the cycle.

In order to ensure that other smart contracts do not access the contracts in the process arbitrarily, we design the data feed management contract, the data feed center contract, and the owner contract with *accessOnly(·)* function. Concretely, this function verifies whether the address of the accessing contract is the address of the specific contract. If the verification is passed, the access proceeds normally. On the contrary, abnormal access will be rejected. The verification process relies on blockchain technology, which ensures the accuracy and reliability of the verification performed on the contracts presented above.
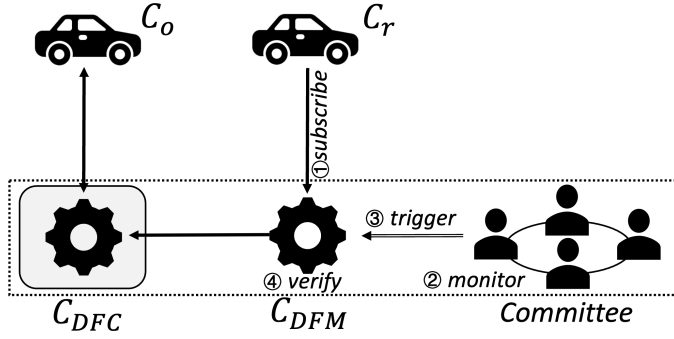
Fig. 8. Subscribe process

Lemma 2. *As long as the owner smart contract is within access range, the requestor smart contract will be able to obtain the desired data.*

Proof. As we described in Section 4.1, we highlight the significance of owner smart contracts during the registration phase and their critical role in the data feed process. In order to ensure that the requestor contract can obtain the desired data, we use function signatures at various stages of the data feed to achieve secure interaction between smart contracts. Whenever the requestor contract calls a contract function via a transaction, the beginning of the transaction data contains the *Keccak256* hash of the function signature, which tells the contract which function to execute. Function signatures act as unique identifiers for functions in a smart contract, ensuring an exact match to the correct function when it is called. This prevents unexpected behavior due to incorrect function names or parameter types and allows contract developers to safely call functions from other contracts without being aware of the exact implementation details.

Lemma 3. *The integrity and security of the data feed management contract C_{DFM} are maintained by the committee if the committee adheres to a threshold t that $t < n/2$, where n is the total number of committee members.*

Proof. The committee plays a crucial role in managing the C_{DFM} contract, which acts as a central hub for coordinating interactions between the various components of the DeFeed protocol. Any modifications to the C_{DFM} contract, including the updating of the C_{DFC} address, are subject to a multi-signature approval process by the committee. This process leverages a (t, n)-secret sharing scheme where $t < n/2$ ensures that no single entity, not even a compromised committee member, can unilaterally alter the contract. This ensures that the core components of the protocol remain secure and under the control of the trusted committee.

The committee also handles the registration and deregistration of data owner contracts (C_o) in the C_{DFC} contract. By implementing a rigorous vetting process, the committee can significantly reduce the risk of including unauthorized or malicious contracts, which is essential for the protocol's resilience against adversarial attacks. Furthermore, the committee actively monitors the DeFeed protocol's activities and performance. It possesses the authority to make data-driven adjustments to the protocol's parameters or governance rules. This committee-based adaptive governance is crucial for maintaining the protocol's security, efficiency, and liveness.

5 IMPLEMENTATION AND EVALUATION

In this section, we implement our protocol mentioned in Section 4 and evaluate each protocol's performance in detail.

5.1 Implementation

We program four key components and implement them with function pool, cache, update, and subscribe in Solidity, the contract-oriented advanced programming language for writing smart contracts. We test all of them in turn over Ethereum's official test network, Sepolia [9].

Next, we implement our basic protocol for realizing data feed. This protocol contains a smart contract C_o that owns the original data, a requestor smart contract C_r that wants to request the data, a data feed management smart contract C_{DFM} for forwarding *request* sending from C_r and a data feed center smart contract C_{DFC} for process the query information from C_o and the forward information from C_{DFM}. In order to achieve the Subscribe function, we design and implement several functions in different contracts respectively. In smart contract C_r, we design and implement the *request(·)* function and the *subscribe(·)* function, respectively. The function *request(·)* aims to enable contract C_r to request contract C_{DFM} with the target contract name. The function *subscribe(·)* allows contract C_r to subscribe to specific contract information from C_{DFM}. The function *forward(·)* and *getSubscribe(·)* are implemented in contract C_{DFM}. The former forward C_r's request to C_{DFC}, and the latter accepts subscriptions and synchronizes subscription information to C_{DFC}. In the contract C_{DFC}, the function *query(·)* and *respond(·)* are designed and implemented. One is used to request data from the data owner C_o, and the other is used to feed data to C_r.

Table 1. Gas Costs in Gas and USD

Function	Gas	USD	Function	Gas	USD
Deploy C_{DFM}	874393	$6.12	Deploy C_{DFC}	1427517	$9.90
Request	143781	$1.00	Update	33241	$0.23
Subscribe	50094	$0.35			
Cache(initial)	221668	$1.55	Cache(subsequent)	60145	$0.42

5.2 Evaluation

We assess the effectiveness of DeFeed by examining the deployment of all smart contracts on the Sepolia test network. The evaluation focuses on the incurred expenses, quantified through fees paid to blockchain miners for executing contract calls. Within the Ethereum ecosystem, the primary metric for assessing costs is *gas*, which encapsulates various factors influencing transaction execution, including computational complexity and storage requirements. Our attention is directed toward the deployment cost of the protocol and the ensuing gas consumption during its operation. We acknowledge the importance of these aspects in the context of our business and academic pursuits and, hence, strive to optimize them for optimal performance.

Approach to evaluation. As done in recent work, our evaluation focuses on measuring the gas consumption of protocols mentioned in Section 4. In TABLE 1, we present a comprehensive overview of the primary functions within the programmed smart contracts that engage with protocol participants at various stages of the protocol. Additionally, we outline the associated costs of these functions in terms of Gas and USD. To compute the costs in USD, we utilize the formula

---

[9]https://sepolia.etherscan.io/

$cost(USD) = cost(Gas) * GasToEther * EtherToUSD$, leveraging the median of $GasToEther$ and $EtherToUSD$ from data of price as documented in Etherscan [10]. Since the historical price of ETH is volatile, we use the median of historical prices to calculate costs. Specifically, $GasToEther$ was determined to be $2.42 * 10^{-8}$ Ether/Gas, while $EtherToUSD$ was established as 289.42 USD/Ether. By applying these values, we accurately compute the costs of the functions in USD based on their respective Gas costs.

Gas cost. As demonstrated in Table 1, the deployment of the two core components of our protocol incurs a little high initial gas costs, 0.87M gas ($6.12) for C_{DFM} and 1.42M gas ($9.9) for C_{DFC}. It is critical to note that these expenses are one-time expenditures. Subsequent uses of the protocol do not require additional gas for these actions, thereby eliminating further deployment costs.

The operational costs of the protocol also warrant discussion. Each smart contract interaction, referred to as a "Request", requires approximately 143,781 gas ($1.00). For contracts subscribing to a target contract, the cost is significantly lower, at only 50,094 gas ($0.35). This subscription mechanism reduces the need for repeated synchronization costs with the target contract, enhancing the protocol's efficiency. Additionally, updating the core contract, C_{DFC}, is relatively inexpensive, requiring only 33,241 gas ($0.23). These features collectively highlight the cost-effectiveness of maintaining the protocol infrastructure.

The Cache function within the protocol introduces further opportunities for gas optimization. While the initial request utilizing the Cache function consumes 221,668 gas ($1.55), reflecting the additional computational resources required for caching, subsequent accesses benefit from a substantial reduction in cost, requiring only 60,145 gas ($0.42). This marked decrease underscores the utility of the Cache function in minimizing long-term operational costs, making it an efficient option for repeated data retrievals within the protocol.
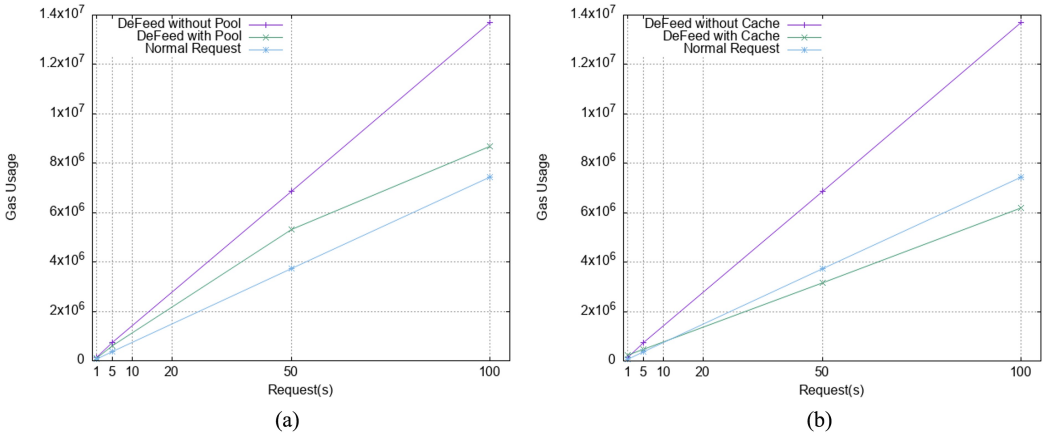

Fig. 9. Gas usage of DeFeed. (a) DeFeed with Pool, and (b) DeFeed with Cache

Gas saving. We compare the gas cost of DeFeed with and without the pool and cache mechanisms, which are introduced in Section 4.1 and Section 4.1, respectively. We design six sets of controlled experiments, each deploying 1, 5, 10, 20, 50, and 100 requesting contracts. These smart contracts send requests to the same data owner contract.

As illustrated in Fig. 9(a), the implementation of the pool mechanism in DeFeed significantly reduces gas consumption. The graph clearly shows three distinct lines: "DeFeed without Pool"

---

[10]https://etherscan.io/

(purple), "DeFeed with Pool" (green), and "Normal Request" (blue). The "Normal Request" mechanism represents the state-of-the-art method, embodying the basic interaction pattern between two contracts. While it shows the lowest gas consumption across all scenarios, with initial usage around 75,000 gas units for a single request, it lacks essential security considerations. This absence of security measures can expose users to vulnerabilities, making it less suitable for applications where trust and safety are paramount.

In contrast, the DeFeed mechanisms are designed with security in mind. Initially, both "DeFeed without Pool" and "DeFeed with Pool" start with similar gas usage for a single request, approximately 149,524 gas units. However, as the number of requests increases, the differences in gas consumption become more pronounced. At 20 requests, the "DeFeed with Pool" mechanism already shows a noticeable advantage, using about 20% less gas than "DeFeed without Pool." By the time we reach 50 requests, the gap widens further, with "DeFeed with Pool" consuming around 5.33 million gas units compared to 6.87 million for "DeFeed without Pool," reflecting a reduction of nearly 22%.

The most striking contrast occurs at 100 requests, where "DeFeed without Pool" consumes approximately 13.7 million gas units, while "DeFeed with Pool" requires only about 8.68 million gas units. This represents a remarkable reduction of around 36%. The graph clearly illustrates that the pool function's efficiency increases with the number of requests, demonstrating how the benefits of this mechanism scale with higher transaction volumes.

Fig. 9(b) introduces the cache mechanism comparison. The graph shows three distinct lines: "DeFeed without Cache" (purple), "DeFeed with Cache" (green), and "Normal Request" (blue). Similar to the pool mechanism, the "Normal Request" method remains the most gas-efficient in terms of raw consumption, but it lacks security, making it unsuitable for secure applications.

For the cache mechanism, "DeFeed without Cache" starts with a gas usage of approximately 149,524 gas units for a single request, while "DeFeed with Cache" starts slightly higher, around 221,668 gas units. As the number of requests increases, the efficiency of the cache mechanism becomes evident. At 50 requests, "DeFeed with Cache" uses approximately 3.17 million gas units compared to 6.87 million for "DeFeed without Cache," indicating a reduction of about 53.8%.

At 100 requests, the efficiency gains are even more pronounced. "DeFeed without Cache" consumes approximately 13.7 million gas units, while "DeFeed with Cache" requires only about 6.18 million gas units, resulting in a substantial reduction of around 54.9%. This demonstrates that the cache mechanism significantly optimizes resource usage at higher volumes of transactions.

In summary, while the "Normal Request" method offers lower gas consumption, it does so at the cost of security, making it less viable for many applications. Our DeFeed protocol, particularly with the pool and cache functions, exhibits substantial efficiency gains, reducing gas usage by significant margins across different levels of request volume compared to "DeFeed without Pool" and "DeFeed without Cache." These performance improvements, clearly visualized in Fig. 9, indicate that the pool and cache mechanisms in DeFeed significantly optimize resource usage while maintaining essential security considerations, making them crucial features for gas optimization in the DeFeed protocol.

Throughput Testing. To evaluate the throughput of our DeFeed protocol, we conduct a series of performance tests using custom scripts. This script simulates multiple transactions and measures the system's ability to process these transactions over time. Our testing methodology focuses on key performance indicators including transactions per second (TPS), average latency, and gas consumption.

We implement the test script using Web3.js to interact with our smart contracts deployed on the Sepolia testnet. The script begins by initializing the connection to the Sepolia network and loading the necessary smart contract ABIs. It then sets a predetermined number of transactions to be executed. Before starting the transactions, the script records the start time of the test. The core
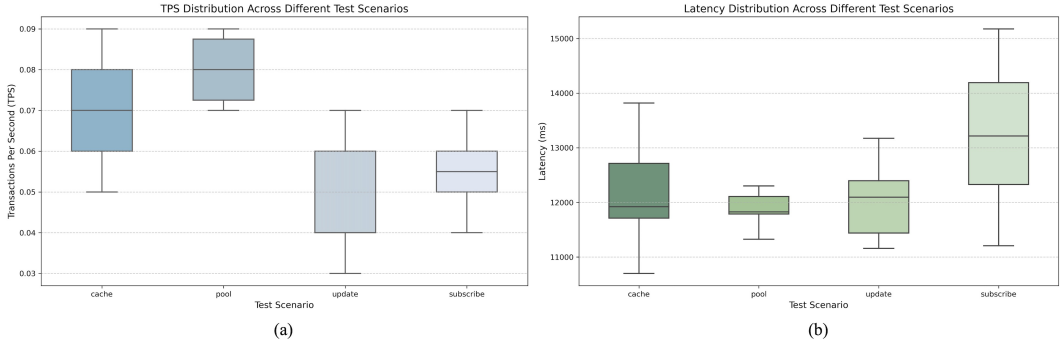
Fig. 10. Performance Metrics of Different DeFeed Mechanisms. (a) Throughput (TPS) distribution across test scenarios, and (b) Latency distribution across test scenarios.

of the script involves executing the specified number of transactions, each calling the *request(·)* function of the C_r contract. Once all transactions are completed, the script records the end time and calculates the total duration of the test. The throughput is computed by dividing the number of successful transactions by the total duration. Additionally, the script calculates average latency and gas consumption for the transactions. To gain insights into network behavior, the script also analyzes block confirmation times.

Our test results show that the DeFeed protocol achieved an average throughput of 0.07 TPS under normal network conditions. The average latency for each transaction was 13881.40 milliseconds, with an average gas consumption of 65532 units per transaction.

To provide a more comprehensive view of the protocol's performance, we also examined the block confirmation times. The average block confirmation time was 13.89 seconds, with a minimum of 12.00 seconds and a maximum of 36.00 seconds. These figures help us understand the variability in transaction processing times due to network conditions and block creation intervals on the Sepolia testnet.

The boxplots in Fig. 10(a) illustrate the throughput performance across four DeFeed mechanisms: cache, pool, update, and subscribe. The pool mechanism demonstrates the highest median TPS (0.08), with a narrow interquartile range (IQR: 0.07–0.09), indicating stable performance due to optimized batch processing. In contrast, the update mechanism exhibits significant variability (median: 0.04, IQR: 0.03–0.07), reflecting inherent inefficiencies in cross-contract state synchronization. The cache strategy shows moderate throughput (median: 0.07) but occasional outliers (e.g., 0.05 and 0.09), likely caused by sporadic network disturbances. The subscribe mechanism achieves the lowest median TPS (0.06), constrained by event-driven overheads and asynchronous callback delays. These results quantitatively validate that batch-oriented designs (pool) outperform event-triggered approaches (subscribe) in high-frequency transaction environments. Fig. 10(b) presents the latency characteristics of the same mechanisms. The cache and pool strategies show tightly clustered latencies (median: 11,798 ms and 11,816 ms, respectively), with 95% of transactions completing within 13,000 ms, demonstrating predictable performance. The update mechanism displays severe right-skewing (median: 12,094 ms; maximum: 15,176 ms), attributable to recursive contract calls and gas contention during state updates. The subscribe mechanism shows the highest variation and the highest average, suggesting that subscribe-related operations are more complex.

It is essential to highlight that these results are specific to the Sepolia testnet environment and are inherently influenced by the network's constraints. Testnets like Sepolia often have lower transaction processing capacity compared to Ethereum's mainnet or other production environments. The

relatively low TPS observed during our tests reflects these limitations, including network congestion and throttled block production intervals, which are designed to simulate realistic conditions but may not fully represent mainnet performance. Consequently, the throughput and latency metrics reported here should not be interpreted as hard limits of the `DeFeed` protocol's capabilities but rather as an indication of its performance under controlled, testnet-specific conditions.

Despite the testnet-imposed constraints, these throughput tests demonstrate that the `DeFeed` protocol can efficiently handle a significant number of requests, making it suitable for applications requiring frequent data feed updates. The relatively low average latency suggests that the protocol is capable of providing near real-time data feeds, a crucial feature for many decentralized applications.

Future work could include more extensive testing under various network conditions, simulating higher concurrency levels, and comparing the performance with other existing oracle solutions to provide a more comprehensive evaluation of `DeFeed`'s capabilities.

6 CONCLUSION

In this paper, we proposed `DeFeed`, a novel and secure protocol for decentralized cross-contract data feed in Web 3.0, specifically designed to enhance the functionality and efficiency of Connected Autonomous Vehicles networks. Our approach leverages blockchain technology and smart contracts to provide a robust and scalable solution for inter-contract communication and data exchange.

We utilized a series of distinct smart contracts to construct our `DeFeed` protocol, which includes two central smart contracts, the data feed management smart contract (C_{DFM}) and the data feed center smart contract (C_{DFC}). First, we introduced a regular data feed process that allows one smart contract to obtain data from another smart contract. Recognizing the inefficiency of handling single requests at a time, we implemented the `Pool` function to enable simultaneous handling of multiple requests, thereby improving overall efficiency. To further optimize resource utilization, we introduced the `Cache` function, which creates a new cache within C_{DFM}. This function stores data obtained from target contracts, allowing subsequent requests for the same data to be fulfilled directly from the cache, thus reducing gas costs and enhancing performance. Additionally, we designed and implemented the `Update` function and `Subscribe` function. The `Update` function allows our protocol to replace the old data center contract C_{DFC} with a new one, facilitating contract updates. The `Subscribe` function enables real-time information changes to be sent from the target contract to the requestor contract, allowing one contract to subscribe to another contract's updates dynamically.

We implemented `DeFeed` over the Ethereum official test network, Sepolia. The results demonstrated that `DeFeed` is economical, convenient to implement, and efficient for facilitating communication and data feed between smart contracts. This makes `DeFeed` a ready-to-be-used protocol for CAV networks within the Web 3.0 ecosystem. Our contributions to the field of autonomous and adaptive systems are significant. We highlighted the role of blockchain technology and smart contracts in achieving secure, decentralized, and efficient communication between entities in autonomous systems. By addressing the limitations of existing oracle-based solutions and focusing on inter-contract data exchange, we provided a practical framework for enhancing the adaptability and interconnectedness of decentralized applications.

The design and implementation of the `DeFeed` protocol showcase the potential of blockchain technology in decentralized autonomous systems and set a new standard for future research and development. `DeFeed`'s innovative features, including the Pool function for processing multiple requests simultaneously, the Cache function for optimizing resource utilization, and the Update and Subscribe functions for real-time data management, collectively enhance the adaptivity and efficiency of decentralized networks. The Pool function, in particular, addresses the challenge of handling multiple data requests efficiently, while the Cache mechanism significantly reduces gas

costs and improves overall system performance. Furthermore, the Update and Subscribe functions ensure that the protocol remains flexible and responsive to changing data needs, crucial for systems like Connected Autonomous Vehicle networks that require real-time decision-making capabilities. These advancements extend beyond mere technical improvements, offering new paradigms for the design and operation of distributed systems. As the Web 3.0 ecosystem evolves, `DeFeed`'s comprehensive approach to inter-contract communication and data exchange positions it as a key enabler for more intelligent and autonomous decentralized applications, thereby shaping the future landscape of digital infrastructure.

While `DeFeed` shows promise, there are areas for future improvement. These include enhancing scalability for high-traffic scenarios, expanding cross-chain compatibility, and further optimizing energy efficiency. Additionally, as the regulatory landscape for blockchain technologies evolves, ensuring compliance across different jurisdictions will be an ongoing consideration. Future work will focus on addressing the identified limitations and exploring new use cases, further solidifying `DeFeed`'s position as a key protocol in the evolving landscape of decentralized technologies.

REFERENCES

[1] Hamda Al-Breiki, Muhammad Habib Ur Rehman, Khaled Salah, and Davor Svetinovic. 2020. Trustworthy blockchain oracles: review, comparison, and open research challenges. *IEEE access* 8 (2020), 85675–85685.

[2] Lennart Ante. 2022. The non-fungible token (NFT) market and its relationship with Bitcoin and Ethereum. *FinTech* 1, 3 (2022), 216–224.

[3] Pedro Antonino, Juliandson Ferreira, Augusto Sampaio, and AW Roscoe. 2022. Specification is Law: Safe Creation and Upgrade of Ethereum Smart Contracts. In *International Conference on Software Engineering and Formal Methods*. Springer, 227–243.

[4] Mohamed Baza, Mahmoud Nabil, Noureddine Lasla, Kemal Fidan, Mohamed Mahmoud, and Mohamed Abdallah. 2019. Blockchain-based firmware update scheme tailored for autonomous vehicles. In *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 1–7.

[5] William E Bodell III, Sajad Meisami, and Yue Duan. 2023. Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1829–1846.

[6] Anne Bumiller, Stéphanie Challita, Benoit Combemale, Olivier Barais, Nicolas Aillery, and Gael Le Lan. 2023. On understanding context modelling for adaptive authentication systems. *ACM Transactions on Autonomous and Adaptive Systems* 18, 1 (2023), 1–35.

[7] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014), 2–1.

[8] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 442–446.

[9] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239.

[10] M Dameron. 2019. Beigepaper: An Ethereum Technical Specification v. 0.8. 5. *Online* (2019).

[11] Yimeng Feng, Guoqiang Mao, Bo Chen, Changle Li, Yilong Hui, Zhigang Xu, and Junliang Chen. 2020. MagMonitor: Vehicle speed estimation and vehicle classification through a magnetic sensor. *IEEE Transactions on Intelligent Transportation Systems* 23, 2 (2020), 1311–1322.

[12] Yu Feng, Emina Torlak, and Rastislav Bodik. 2019. Precise attack synthesis for smart contracts. *arXiv preprint arXiv:1902.06067* (2019).

[13] Christof Ferreira Torres, Hugo Jonker, and Radu State. 2022. Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*. 115–128.

[14] Uriel Fiege, Amos Fiat, and Adi Shamir. 1987. Zero knowledge proofs of identity. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*. 210–217.

[15] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. {ETHBMC}: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*. 2757–2774.

[16] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[17] Juan Guarnizo and Pawel Szalachowski. 2019. PDFS: practical data feed service for smart contracts. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*. Springer, 767–789.

[18] Rateb Jabbar, Noora Fetais, Mohamed Kharbeche, Moez Krichen, Kamel Barkaoui, and Mohammed Shinoy. 2021. Blockchain for the Internet of Vehicles: How to use blockchain to secure vehicle-to-everything (V2X) communication and payment? *IEEE Sensors Journal* 21, 14 (2021), 15807–15823.

[19] Saurabh Jain, Neelu Jyothi Ahuja, P Srikanth, Kishor Vinayak Bhadane, Bharathram Nagaiah, Adarsh Kumar, and Charalambos Konstantinou. 2021. Blockchain and autonomous vehicles: Recent advances and future directions. *IEEE Access* 9 (2021), 130264–130328.

[20] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 259–269.

[21] Tigang Jiang, Hua Fang, and Honggang Wang. 2018. Blockchain-based internet of vehicles: Distributed network architecture and performance analysis. *IEEE Internet of Things Journal* 6, 3 (2018), 4640–4649.

[22] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 839–858.

[23] Chao Li and Balaji Palanisamy. 2020. Eventwarden: A decentralized event-driven proxy service for outsourcing arbitrary transactions in ethereum-like blockchains. In *2020 IEEE International Conference on Web Services (ICWS)*. IEEE, 9–16.

[24] Chao Li and Balaji Palanisamy. 2021. Silentdelivery: Practical timed-delivery of private information using smart contracts. *IEEE Transactions on Services Computing* 15, 6 (2021), 3528–3540.

[25] Chao Li and Balaji Palanisamy. 2024. T-Watch: Towards Timed Execution of Private Transaction in Blockchains. *IEEE Transactions on Services Computing* (2024).

[26] Chao Li, Balaji Palanisamy, Runhua Xu, Li Duan, Jiqiang Liu, and Wei Wang. 2023. How hard is takeover in dpos blockchains? understanding the security of coin-based voting governance. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 150–164.

[27] Chao Li, Balaji Palanisamy, Runhua Xu, Jian Wang, and Jiqiang Liu. 2020. Nf-crowd: Nearly-free blockchain-based crowdsourcing. In *2020 International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 41–50.

[28] Nianyu Li, Mingyue Zhang, Jialong Li, Sridhar Adepu, Eunsuk Kang, and Zhi Jin. 2024. A Game-Theoretical Self-Adaptation Framework for Securing Software-Intensive Systems. *ACM Transactions on Autonomous and Adaptive Systems* 19, 2 (2024), 1–49.

[29] Bowen Liu, Hao Tian, Zhijie Shen, Yueyue Xu, and Wanchun Dou. 2024. A Consortium Blockchain-Based Edge Task Offloading Method for Connected Autonomous Vehicles. *ACM Transactions on Autonomous and Adaptive Systems* (2024).

[30] Sin Kuang Lo, Xiwei Xu, Mark Staples, and Lina Yao. 2020. Reliability analysis for blockchain oracles. *Computers & Electrical Engineering* 83 (2020), 106582.

[31] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.

[32] Swati Megha, Hamza Salem, Enes Ayan, and Manuel Mazzara. 2020. A survey of blockchain solutions for autonomous vehicles ecosystems. In *Journal of Physics: Conference Series*, Vol. 1694. IOP Publishing, 012024.

[33] Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei, and Chen Qijun. 2017. A review on consensus algorithm of blockchain. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2567–2572.

[34] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review* (2008).

[35] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

[36] Geetanjali Rathee, Ashutosh Sharma, Razi Iqbal, Moayad Aloqaily, Naveen Jaglan, and Rajiv Kumar. 2019. A blockchain framework for securing connected and autonomous vehicles. *Sensors* 19, 14 (2019), 3165.

[37] Wei Ren, Xutao Wan, and Pengcheng Gan. 2021. A double-blockchain solution for agricultural sampled data security in Internet of Things network. *Future Generation Computer Systems* 117 (2021), 453–461.

[38] Hubert Ritzdorf, Karl Wüst, Arthur Gervais, Guillaume Felley, and Srdjan Capkun. 2018. TLS-N: Non-repudiation over TLS Enabling Ubiquitous Content Signing. http://dx.doi.org/10.14722/ndss.2018.23272. In *Network and Distributed Systems Security (NDSS) Symposium 2018*. 18–21.

[39] Dongxian Shi, Xiaoqing Wang, Ming Xu, Liang Kou, and Hongbing Cheng. 2023. RESS: A reliable and efficient storage scheme for bitcoin blockchain based on raptor code. *Chinese Journal of Electronics* 32, 3 (2023), 577–586.

[40] Pranav Kumar Singh, Roshan Singh, Sunit Kumar Nandi, Kayhan Zrar Ghafoor, Danda B Rawat, and Sukumar Nandi. 2020. Blockchain-based adaptive trust management in internet of vehicles using smart contract. *IEEE Transactions on*

Intelligent Transportation Systems 22, 6 (2020), 3616–3630.

[41] Jian Su and Mengnan Jiang. 2023. A hybrid entropy and blockchain approach for network security defense in SDN-based IIoT. *Chinese Journal of Electronics* 32, 3 (2023), 531–541.

[42] Nick Szabo et al. 1994. Smart contracts.

[43] Hao Wang, Zhe Liu, Chunpeng Ge, Kouichi Sakurai, and Chunhua Su. 2022. A privacy-preserving data feed scheme for smart contracts. *IEICE TRANSACTIONS on Information and Systems* 105, 2 (2022), 195–204.

[44] Shuai Wang, Liwei Ouyang, Yong Yuan, Xiaochun Ni, Xuan Han, and Fei-Yue Wang. 2019. Blockchain-enabled smart contracts: architecture, applications, and future trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 49, 11 (2019), 2266–2277.

[45] Yingsen Wang, Leiming Yuan, Weihan Jiao, Yan Qiang, Juanjuan Zhao, Qianqian Yang, and Keqin Li. 2023. A fast and secured vehicle-to-vehicle energy trading based on blockchain consensus in the internet of electric vehicles. *IEEE Transactions on Vehicular Technology* 72, 6 (2023), 7827–7843.

[46] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[47] Xiaolong Xu, Yi Chen, Yuan Yuan, Tao Huang, Xuyun Zhang, and Lianyong Qi. 2020. Blockchain-based cloudlet management for multimedia workflow in mobile cloud computing. *Multimedia Tools and Applications* 79 (2020), 9819–9844.

[48] Xiaolong Xu, Ji Gu, Hanzhi Yan, Wentao Liu, Lianyong Qi, and Xiaokang Zhou. 2022. Reputation-aware supplier assessment for blockchain-enabled supply chain in industry 4.0. *IEEE Transactions on Industrial Informatics* 19, 4 (2022), 5485–5494.

[49] Xiaolong Xu, Dawei Zhu, Xiaoxian Yang, Shuo Wang, Lianyong Qi, and Wanchun Dou. 2021. Concurrent practical byzantine fault tolerance for integration of blockchain and supply chain. *ACM Transactions on Internet Technology (TOIT)* 21, 1 (2021), 1–17.

[50] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. 2022. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2022).

[51] Kunwei Yang, Bo Yang, Tao Wang, and Yanwei Zhou. 2023. Zero-cerd: a self-blindable anonymous authentication system based on blockchain. *Chinese Journal of Electronics* 32, 3 (2023), 587–596.

[52] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 270–282.