Blockchain-based System Towards Data Security Against Smart Contract Vulnerabilities: Electronic Toll Collection Context

Olfa Ben Rhaiem[1]*, Marwa Amara[2], Radhia Zaghdoud[3], Lamia Chaari[4], Maha Metab Alshammari[5]
Department of Computer Science, College of Science, Northern Border University, Arar, Saudi Arabia[1,2,3,5]
Digital Research Center of SFax (CRNS), SM@RTS (Laboratory of Signals, Systems, Artificial Intelligence and Networks), Sfax, Tunisia[1,4]

***Abstract*—Electronic Toll Collection (ETC) systems have been proposed as a replacement for traditional toll booths, where vehicles are required to queue to make payments, particularly during holiday period. Thus, the primary advantage of ETC is improved traffic efficiency. However, existing ETC systems lack the security necessary to protect vehicle information privacy and prevent fund theft. As a result, automatic payments become inefficient and susceptible to attacks, such as Reentrancy attack. In this paper, we utilize Ethereum blockchain and smart contracts as the automatic payment method. The biggest challenges are to authenticate the vehicle data, automatically deducts fees from the user's wallet and protects against smart contract Reentrancy Attack without leaking distance information. To address these challenges, we propose an end-to-end Verification algorithms at both entry and exit toll points that corporate measures to protect distance-related information from potential leaks. The proposed system's performance was evaluated on a private blockchain. Results demonstrate that our approach enhances transaction security and ensures accurate payment processing.**

***Keywords*—Blockchain; Ethereum; smart contracts; Reentrancy Attacks; security; ETC**

I. Introduction

The number of vehicles on the highway increases rapidly day by day. Vehicles passing through toll plaza need to pay the toll-tax amount (TA). In this case, vehicles are sent to the waiting queue willing to pay resulting in delay in time, traffic congestion and more fuel consumption.

Recently, Electronic Toll Collection (ETC) systems have been proposed to replace the traditional charging mode in the highway stations and address the aforementioned issues. Particularly, the main advantage of ETC is to improve traffic efficiency. In fact, ETC systems automatically collect the usage fees without requiring any action or stopping by the driver.

But, in IOV, vehicles are equipped with sensors named as the On Board Unit (OBU) [12]. These sensors collects and exchange information from stationery Road Side Units (RSU) and electronic toll collection systems. In the way of centralized systems, security and effectiveness of data exchanged wit ETC makes the communication difficult. Particularly, exchanged information involves critical information (e.g., location which is used to compute the traveled distance) is highly susceptible to spoofing attacks. which means that the amount of fees is not accurately calculated. This problem becomes more challenging when the distance information (based on location) is required to have an accurate amount of fees.

Recently, blockchain technology [3], which is applied in different fields, is defined as a new way to enhance security. Blockchain combines special features such as decentralized structure, consensus algorithm, smart contracting, and asymmetric encryption to ensure network security. Consequently, data is protected and cannot be stolen by hackers.

Blockchain technology has many other applications that go beyond digital currencies. In fact, Bitcoin is one of several applications that uses blockchain technology. The second generation of blockchain technology represented by Ethereum [9] was launched in July 2015. Ethereum*, is an open and fully decentralized platform enabling a new paradigm of computing Decentralized Applications (DApps) running on top of blockchains. Ethereum uses smart contracts which allow users to set and retrieve data from the Ethereum network. Smart contracts facilitate to exchange of money and any data values. Smart contracts [10], [11] cannot be updated or modified after their deployment on a blockchain network. However, despite all its advantages, smart contracts are not fully secured and faces challenges of various attacks. In fact, an important risk is that hackers use another technique called Reentrancy attack which is one of the most destructive attacks that makes transactions not secured. Reentrancy attacks are most often associated with Ethereum Blockchain. Thus, using blockchain technology for Electronic Toll Collection in internet of vehicle (IOV) is an important aspect that does not guarantee the data transmission in a secure way.

Based on these issues, in this paper we have proposed a decentralized application to secure smart contracts, for Electronic Toll system (ETC), using Ethereum blockchain that protects transactions from malicious hackers. The proposed system preserves privacy for vehicle's information and ensure a correct service of payments.

The proposed system offers the following benefits:

- The proposed ETC system would help reduce traffic congestion and wait times at toll plazas. This would result in improved traffic flow and reduced travel time for drivers.

*Corresponding authors.

*last accessed on 01-10-2019. [Online]. Available: https://www.ethereum.org/

- Electronic Toll Collection system is based on Blockchain technology which is executed without the need of third party.
- Smart contracts are used to provide the security of the exchanged information on the ETC. In fact, since data are stored in a decentralized system, the chance of modifying data is very difficult.
- The proposed blockchain-based ETC system secures smart contract from attacks (e.g., Reentrancy Attack is exploited to steal funds from smart contracts).
- The proposed ETC system accurately calculate toll fees based on the traveled distance and automatically deduct the exact toll fees from the user's wallet.
- Implementing an end-to-end Verification algorithms at both entry and exit toll points that corporate measures to protect distance-related information from potential leak.
- Creates a verification system of vehicle's location to guarantee a correct payments service.
- Using the protecting safeMath library provided by Openzeppline module to ensure the security of smart contracts.

The remaining part of this paper is organized as follows: Section II reviews the most related blockchain-based approaches. Section III gives a better understanding of the basics of blockchain framework. Section IV elaborates the proposed system model and designed solution, which includes an end-to-end verification algorithms at both entry and exit tolls. Section V describes the implementation of the Decentralized application and evaluate it. Section VI concludes the paper.

II. Related Works

Electronic Toll Collection Systems (ETC) are an important part of the intelligent transportation system (ITS). Several works are proposed in the literature (e.g. [1], [2], [3], [5], [13]) to analyze the security issues and challenges of ETC systems. Some blockchain-based ETC approaches provide security for vehicle's information and guarantee an accurate automatic payment services. However, one biggest challenge related to smart contract security must be addressed. Particularly, reentrancy attack is the most destructive attack in Solidity smart contract to steal funds.

This section discusses the existing blockchain-based ETC approaches and highlights the concept of reentrancy attack which is one of the most destructive attacks in solidity smart contract.

A. Blockchain-based ETC Approaches

Authors in [1] showed that current ETC systems are not efficient and have vehicle fee evasion complications. To solve issues, authors proposed a data management method to improve the security of the data transmission process, without affecting the system performance. This solution is based on the alliance chain. Although this system reduces the number of illegal acts and improves data security, it has some deficiencies including not being completely decentralized.

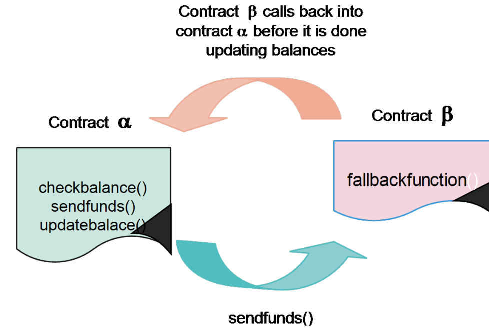

Fig. 1. Reentrancy Attack.

In the same context and in order to collect the Toll-Tax Amount (TA) without slowing down a vehicle's speed at the toll plazas; authors in [2], have combined blockchain with the Electronic Toll collections systems and proposed a new system named as Blockchain-based Automated Toll-Tax Collection System (BATCS). In fact, the system uses smart contracts to authenticate the vehicle data and collect TA automatically at toll plazas. Less fuel consumption and more time-saving for a vehicle are the main benefits of the BATCS.

Considering the immutability of smart contracts, authors in [4] proposed an authentic and secure automatic payment system. This system verifies the location of the specific vehicle by the other vehicle owners. The experimental results are based on the gas consumption of all the smart contract and ensures that the vehicle has traveled to the correct location.

In [7], authors have proposed two scenarios of Blockchain based secure payment scheme in VANET: i) park toll management system and; ii) electronic toll collection. In this work, only RSU takes part in the consensus and all transaction run in the smart contract automatically. Also, it is able to mitigate security and privacy requirements.

To address the issues in relation to data storage, trustworthiness, and transparency, authors in [8] proposed a blockchain-based ETC system named as EdgeTC. Unlike most other blockchain-based ETC platforms that use Proof of work (PoW) or Proff of stack (PoS), this system uses PBFT to achieve relatively faster performance.

B. Smart Contract Hacking (Reentrancy Attack)

In this section we will understand how Reentrancy Attack works. A Reentrancy Attack is a type of serious attack that affects smart contracts on blockchain platforms, such as Ethereum. In general, a Reentrancy Attack [14] can allow an external party to enter the contract and eventually drain the funds from that contract. This attack happens if a contract fails to update its state before sending funds, this will create a chance for the attacker to drain the contract's funds. In fact, if a contract calls another contract β, the Ethereum protocol allows β to call back to any public or external method θ of α in the same transaction before even finishing the original invocation. An attack happens when β reenters α in an inconsistent state before α gets the chance to update its internal state in the original call (see Fig. 1).

Based on the example depicted in Fig. 1, reentrancy is a serious vulnerability that is quite dangerous because this can drain out the entire funds of the contract. Thus, Reentrancy Attacks lead to steel innocence people's money. Several works have been proposed [1] [2] [4] [6] to secure smart contracts and ensure that all transactions in Ethereum blockchain are not subject to Reentrancy Attacks.

Based on the work comparison discussed in Table I, we conclude that none of ETC-based blockchain systems help to secure transactions against both vehicle's information (especially location spoofing) and Reentrancy Attacks related to smart contracts on Ethereum Blockchain. In fact, protecting the location of vehicle to ensure a correct payment service is not enough since smart contracts are susceptible to several attacks such as Reentrancy Attack.

For that purpose, we have proposed a novel decentralized application based on blockchain technology for ETC systems using Ethereum blockchain. First, this proposal aims to secure the vehicle's information (basically location) to get an accurate amount of fees. Then, this work aims to get a smart contract that is immune to Reentrancy Attacks.

III. Blockchain Framework

This section presents a background information related to blockchain data model and Ethereum framework. We start by outlining the basic structure of Ethereum block header and data. Then, we present the different steps of the Ethereum blockchain framework for the transaction cycle.

A. Blockchain Data Model

Blockchain is a completely new way to share data. It allows us to make transactions in a way that are more secure and more transparent. With blockchain data isn't held in a centralized database. It is shared with everyone and verified by people in the network. Information is secured using cryptography so that criminals can't come and steal stored data. This makes this type of data breach nearly impossible.

Basically, blockchain is a shared database that contains a list of transactions, and these transactions are made between the users who become part of this network. The transaction is sent out to a network of users and the goal of this network is to take all transactions and group it with other transactions into a block. Once enough transactions are collected the block if full and ready to be permanently added to the blockchain.

To give more control about this list of transactions, the blockchain is split up into smaller sections known as blocks. Information is held in part of the block known as the block header. This header details the structure of the data inside the block: the hash of the previous block, the timestamp the block was made, the Merkel root and the nonce all sit inside the block's header as shown in Fig. 2. The body of the block contains a set of transactions.

- Previous Block Hash: it is a block hash for the block that comes directly before the given block in the chain. Having this connection links, the blocks together by allowing to always know what block comes before and after any block on the chain. This forms the basis of the entire blockchain.
- Timestamp: shows that the blocks are connected in a chronological order. It marks the time for each transaction on the blockchain. Simply put, the time proves when and what has happened on the blockchain and its tamper-proof. Timestamp plays to role of a notary and it is more credible than a traditional one; because no body can alter the information on the blockchain.
- Merkle Root: is the Hash that represents every transaction inside the block. To get the Merkel Root, pair of the transactions within the block are repeatedly hashed together. Each pair results in a single hash. Then a hash of two pairs of transactions is again hashed together; over and over again until we left with a single hash value. Given that final hash value is known as Merkel root, hashing is reversed to reconstruct the entire set of transactions from the original block.
- Nonce: Is an arbitrary number that can only be used once. When creating a hash for a block, not just any value will work. The system requests a very specific hash value that starts with a certain number of zeros. These extra constraints make the hash more difficult to find. To find that value, blocks data are combined with the nonce to generate the correct hash value. Computers guess this nonce over and over again until finally come up with the value that gives a hash that meets the constraints.
  As we can see in Fig. 2, each block contains its own hash plus the hash of the previous block. These hash values chain the blocks together in order form the most recent block made all the way to the first block ever created. The fact that these blocks are connected by hash values gives them some interesting qualities. We know that if we change the data on a block, it will create a new hash value for that block. That will invalidate the block and since the hash for the block changes, it also changes the hash for this block that exists on the next block. This change of hash runs all the way down the set of blocks effectively breaking the entire chain as shown in Fig. 3.

B. Ethereum Blockchain Framework

In this paper, we consider Ethereum, which is one of the earliest and most widely deployed smart contract platforms. Ethereum has several advantages such as, flexibility, completeness, and availability of its development tools. Moreover, it designs a virtual machine specifically for running smart contracts named as Ethereum Virtual Machine (EVM) [15]. Solidity is one of the programming languages that is specifically designed for smart contracts by the Ethereum Team.

Ethereum provides two types of accounts: Externally Owned Account (EOA) which is controlled by individuals through the use of private key; the second one is contract account controlled by smart contract and it don't require the use of any private key. Both accounts have a specific address, which is basically the account identifier, an ETH balance and can send transactions to the Ethereum network. The main difference is that contract accounts can't initiate transactions on their own, they first need to be triggered by a user vehicle

TABLE I. ETC-BASED BLOCKCHAIN APPROACHES

Citation-year	Blockchain technology	consensus algorithms	Advantages	Security issues
[1]-2021	Blockchain framework (Hyperledger Fabric)	Proof of Stake (PoS)	The system effectively reduces the number of illegal acts, -solve the problem of escaping fees -Improves the security of the data.	-Amount of fees is susceptible to Reentrancy Attacks. -Does not guarantee that the amount of fees is accurately calculated.
[2]-2022	Ethereum	Proof of Stake (PoS)	-collect the Toll-Tax Amount automatically at toll plaza. -uses smart contract to authenticate vehicles. - Less fuel consumption.	Does not guarantee that the amount of fees is accurately calculated. Amount of fees is susceptible to Reentrancy Attacks
[4]-2022	Ethereum	Proof of Stake (PoS)	- Ensures that the vehicle has traveled to the correct location. -Authentic and secure automatic payment system.	- Reentrancy vulnerability cannot be detected accurately
[6]-2022	private Ethereum	N/A	-new cryptographic technique zk-GSigproof, which preserves privacy while guaranteeing correct payment amount. -Vehicloak is highly efficient in processing payments on the blockchain	- transaction gas and on-chain workload are significantly high
[7]-2020	Ethereum	Proof of Stake (PoS)	-cryptography building blocks protects the security and privacy of vehicle accounts. -Only RSUs participate in the consensus mechanism, and vehicles can obtain data through RSU, which ensures the fast synchronization of data stored by all entities in the blockchan	-no comparative analysis with similar payment approaches in VANETs
[8]-2021	Ethereum	Practical Byzantine Fault Tolerance (PBFT)	- Blockchain-based ETC system based on PBFT	- PBFT is not suitable for large scale network - Reentrancy vulnerability cannot be detected accurately

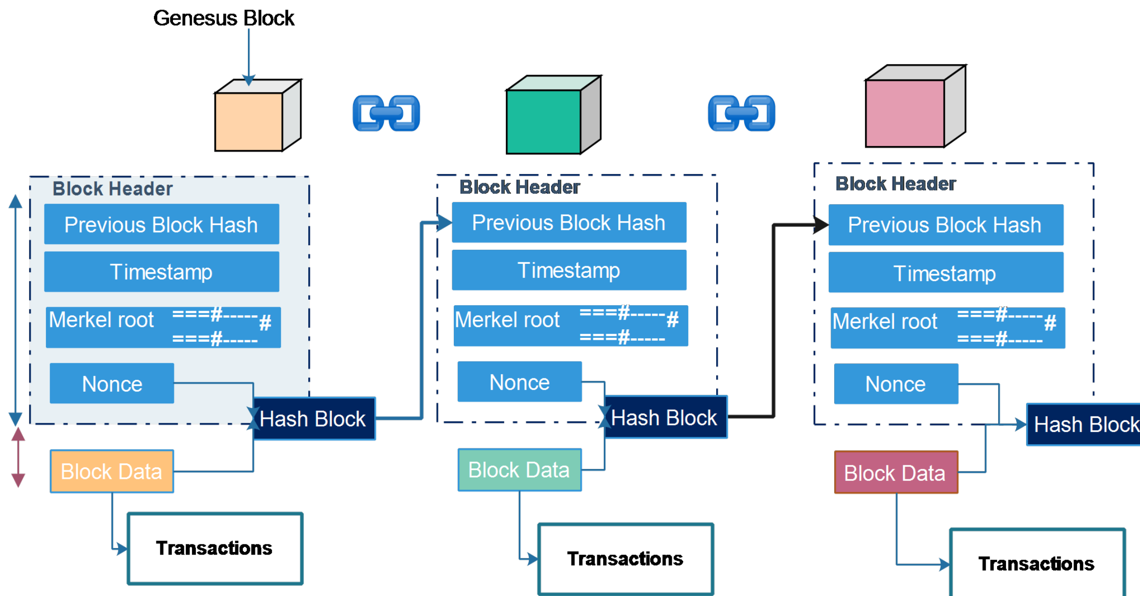

Fig. 2. Block structure.

with an EOA and that trigger can essentially cause the contract account to execute actions or even creating new contracts (see Fig. 4).

The framework depicted in Fig. 5 is an effective way to understand Ethereum Blockchain.

The wallet contains the wallet address shared with others used to receive crypto currencies like Ethereum. Thus, we denoted by $User_1$ the user who wishes to send transaction T_i to $User_2$, using his wallet address. In fact, the wallet is responsible for storing the private Key of the address, denoted as Pr_{key}, sending transaction and showing the balance.The first step in using a wallet is to generate the wallet address. The wallet first generates a random series of 12 words known as a mnemonic phrase, which will be used to generate a private key. This private key is used to send transactions and generates the public key, denoted as Pb_{key} . Finally, using Pb_{key} the wallet address is generated as shown in Eq. (1).

$$wallet - address = Keccak - 256(pb_{key}) \quad (1)$$

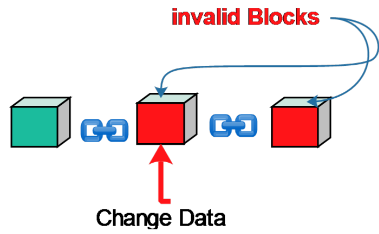

Fig. 3. Invalid blocks.

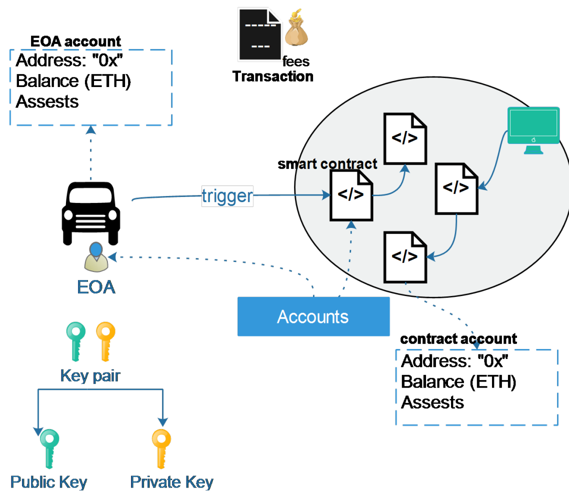

Fig. 4. EOA and contract accounts.

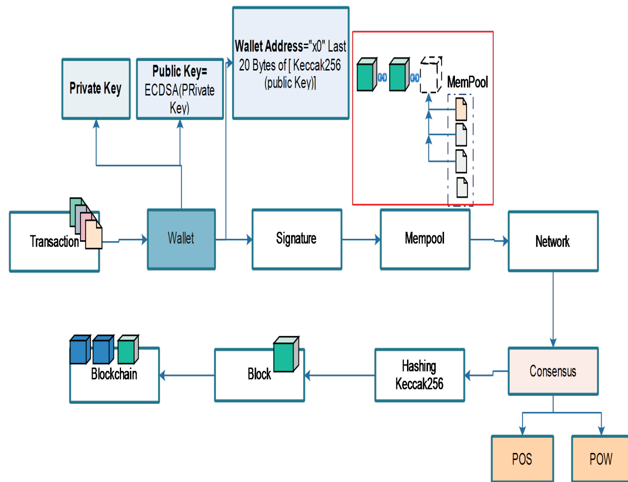

Fig. 5. Blockchain framework.

Where:

Keccak-256 is the hashing algorithm used to create the wallet address.

Once $User_1$ sends the transaction using the wallet address, the hash of the transaction object using the Keccak-256 hashing algorithm is calculated as shown in Eq. 2.

$$T_i^{hash} = Keccak - 256(T_i^{Data}) \tag{2}$$

Where:

T_i^{hash} is the hash value of the ith transaction data, denoted by T_i^{Data}

Then, the sender signs the transaction using their private key and the ECDSA [] algorithm. The signature is generated by creating a digital signature on the hash of the transaction using the sender's private key (see Eq.(3)). It is then added to the transaction object along with the sender's public key and any other required information.

$$signature = ECDSA(Pr_{key}, Keccak - 256(T_i^{hash})) \tag{3}$$

Where:

- ECDSA is the Elliptic Curve Digital Signature Algorithm
- Pb_{key} is the sender's private key
- Keccak-256 is the hashing algorithm used to calculate the transaction hash
- T_i^{hash} is the hash of the transaction.
- $signature$ is the resulting digital signature on the transaction

Before becoming a part of the network, and eventually the blockchain, the transaction is held in the memory pool, which is a temporary storage where unconfirmed transactions are held while they await inclusion in a block by a miner.

Transactions in the mempool are ordered by their gas price. Miners favour transactions with greater gas prices.

If a transaction is stuck in the MemPool for too long (i.e. considered as invalid or gas fee too low for a miner to ever pick it up) it will be rejected by the network and removed from the mempool.

When a miner is ready to mine a new block, they will typically select the highest gas price transactions from the mempool to include in the block. The remaining transactions in the mempool will continue to wait for inclusion in the next block. Thus, the transaction waiting in the MemPool is in hopes of being validated. Then they can leave and permanently be added to the blockchain.

IV. Proposed System

In this section, we describe the architecture of the proposed end-to-end blockchain-based ETC system at both entry and exit toll. The proposed system include the following main steps: i) authentication process that includes two sub-steps: the information gathering and vehicle registration; ii) distance calculation; and iii) payment process using a secure smart contract.

Particularly, the general working process is described as follow:

1) Identify the unique vehicle-ID
2) Verify the authenticity of each transaction
3) Check if the vehicle is registered
4) Calculate the distance
5) Calculate the toll amount

TABLE II. ABBREVIATIONS AND SYMBOLS

Symbols	Description
St_1	Entry station in an ETC system
St_2	Exit station in an ETC system
RSU	Road Side Unit
V_e	user's vehicle
RFID	Radio Frequency Identification
M_n	Blockchain Miners
S	Servers that control the overall ETC system functionality
$VehId$	Vehcile's Identity
L	The geographic coordinates associated with each vehicle.
$user_{addr}$	the driver's address

6) Deduct the amount from the driver's account.

These steps are involved at exit/entrance toll stations.

A. System Architecture

The global architecture of the proposed blockchain-based ETC system is shown in Fig. 6. It requires five key entities:

- Two ETC stations (St_i; $i = 1, 2$); where St_1 represents the entry ETC station and St_2 represents the exit ETC station. End-to-end security algorithms are implemented at exit/entrance toll stations. In fact, St_1 and St_2 are equipped with sensors to read RFID (A Radio Frequency Identification) or other forms of electronic identification that are installed on the vehicles.
- vehicles (V_e) passing through tollgate are equipped with an RFID tag or other electronic identification. When the vehicle passes through the exit/entrance toll station, RSU (Road side Unit) reads its RFID information through wireless communication.
- The system is equipped with servers (denoted as S) which manage all the driver's transactions at exit/entrance toll stations.
- Road side Units (RSUs): The RSUs will be installed along the toll road and especially at the entry and exit gates. These units would communicate with the servers and the vehicles using blockchain technology. RSUs would be responsible for collecting and transmitting data related to the vehicles passing through the toll plaza.
- Blockchain miners (Mn) would be responsible for validating and processing transactions on the blockchain.

In this work, we assume that Blockchain-based ETC system is implemented on highways to facilitate faster and more efficient toll payments for vehicles traveling long distance without interruption. Moreover, the Blockchain-based ETC system is designed to be interoperable with other toll systems across the country to ensure seamless travel for vehicles traveling long distance.

In Table II, we present a list of abbreviations and symbols used throughout the paper. Each abbreviation or symbol is defined alongside its corresponding meaning or representation.

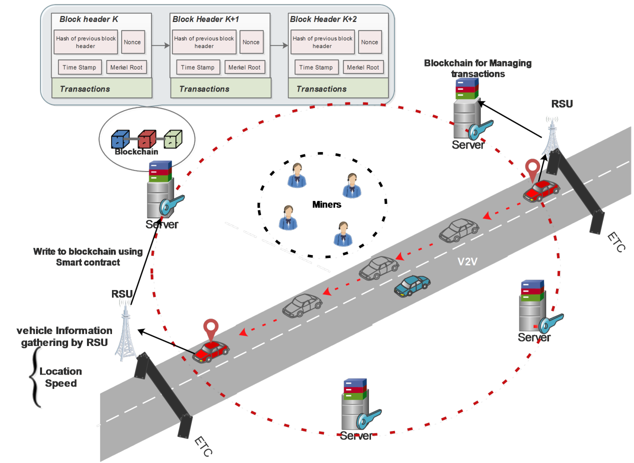

Fig. 6. Global system architecture.

B. Step 1: Authentication Process

In this sub-section we give details about two sub-steps included in the authentication process (i.e., information gathering and vehicle registration) of the vehicle passing through the entrance toll.

When a driver wants to register their vehicle in the blockchain-based ETC system, they would initiate a registration request by submitting their vehicle information to the smart contract deployed in the network.

In fact, considering a vehicle (denoted by V_e) in which a Radio Frequency Identification (RFID) is installed. When V_e passes through the entry toll (St_1), its RFID tag is scanned by the Road-Side Unit (RSU) to retrieve the vehicle's identity ($vehID$). Meanwhile, it is essential to ensure the validation of data and prevent the recording of duplicate or invalid entries. This is achieved thorough two main sub-processes: i) data validation; and ii) Duplicate Check.

Let V represents the vehicle's information. The validation process verifies the correctness of V. Note that $V_{\text{validated}}$ represents the validity of the vehicle's information. If V are not valid, the ETC will restrict this vehicle from accessing the highway and the status of the current vehicle will be set as illegal.

Once V has been validated, the duplicate check sub-process is initiated to prevent registering the same vehicle multiple times. This involves comparing $V_{\text{validated}}$ with the existing records in the blockchain-based ETC system, denoted as R. The comparison is denoted as $V_{\text{validated}} \cap R$. If $V_{\text{validated}} \cap R = \emptyset$, it means that the vehicle is not registered previously. Once the double check and validation of the vehicle data (V) have been successfully completed, the registration process is initiated.

In fact, the vehicle user initiates the registration process by sending a request to the blockchain system. The vehicle communicate with the RSU device in the ETC channel by sending its information (e.g., vehicle's identity ($vehID$), user address ($user_{addr}$) and its current location (L)).

Subsequently, the RSU generates a transaction that interacts with the smart contract on the Ethereum blockchain to store

the transaction data. This smart contract is designed to store and manage vehicle registration data. The RSU pays a small fee in the form of gas, which is used to incentivize miners to include the transaction in the blockchain.

The transaction of the ith vehicle including the function call is then created to be recorded within the blockchain-based ETC system (see Eq. 5).

$$T_i^{hash} = registerUser(user_{addr}, vehID, L)) \quad (4)$$

Where $user_{addr}$ is the account address of the vehicle being registered.

The created transaction contains the vehicle's ID, location data, the user's address as well as the current time, gas price, etc). This transaction is, then, hashed using a secure hash function such as Keccak-256. This generates a unique fixed-size output that serves as a digital fingerprint of the transaction data. Then a digital signature using the V_e's private key is sent to the ETC system along with the vehicle's registration details. This proves that the transaction could only have come from the specific vehicle and was not sent fraudulently.

The smart contract validates the registration request by checking if the vehicle information provided is valid. Once the registration request is validated, the transaction is broadcast to the Ethereum network, where miners execute the smart contract to verify the uploaded information.

According to the verification process; if the block including the transaction is approved, it is added to the blockchain as the latest block. In this case, for for each registered vehicle, a unique vehicle identifier, denoted as UV_{ID}, is generated. This identifier serves as a distinct reference for the vehicle within the blockchain-based ETC-system, allowing for efficient tracking and management of registered vehicles.

To prevent false information attacks, the RSU receives a confirmation message, from the Ethereum network, which indicates that the vehicle's information has been successfully registered in the Blockchain-based ETC system. Meanwhile, a unique Vehicle's ID, denoted as UV_{ID} is generated for each vehicle registered in the blockchain. The whole authentication process of each vehicle passing through the entrance toll is resumed in Algorithm 1.

During the registration process some constraints must be checked:

- Firstly, it checks that the contract is deployed, and contract address is generated.
- Secondly, it confirms that a vehicle cannot be registered before owner is registered.
- Finally, it confirms that an owner address cannot be registered as a vehicle user address.

C. Data Collection and Distance Calculation Methodology

This subsection outlines the methodology used for distance calculation and location verification at the exit-ETC, as shown in Fig. 7.

Algorithm 1 Registration Process: Entrance ETC

1: **Input:** $\mathbf{vehID}, \mathbf{L}, \mathbf{user_{addr}}, \mathbf{Pr_{key}}$ ▷ the vehicle's information denoted as V
2: **Output:** Message
3: **if** V is valid **then**
4: $V_{status} \leftarrow$ Validated
5: **if** $vehID, Pr_{key}, L, user_{addr}$ NOT stored in the Blockchain **then**
6: $T_i \leftarrow$ registerUser(vehID, L, user$_{addr}$)
7: $Value \leftarrow$ Verify(T_i)
8: **if** ($Value$==true) **then**
9: $H_i \leftarrow$ Hash(T_i)
10: $S_i \leftarrow$ Sign(H_i, Pr_{key})
11: $tab \leftarrow$ AddMempool(T_i)
12: Broadcast(T_i) ▷ transaction is broadcasted to the Ethereum network
13: $Val \leftarrow$ Validate(T_i) ▷ to ensure that it meets certain criteria.
14: **if** (Val) **then**
15: Mining(T_i) ▷ Miners verify the transactions and create new blocks.
16: $VehStatus \leftarrow$ Registered
17: UV_{ID} is generated
18: **end if**
19: **end if**
20: **else**
21: $Message \leftarrow$ Vehicle already registered
22: **end if**
23: **else**
24: $VehStatus \leftarrow$ illegal
25: $V_{Status} \leftarrow$ invalid
26: **end if**

When a vehicle V_e gets onto the highway and passes through the exit-ETC; ETC identifies the vehicle's information to get parameters required for the identification of the current vehicle in the ETC-blockchain through RSUs. This identification process is done in the purpose of automatically deducting the toll amount from the vehicle owner's account based on the vehicle registration information stored in the smart contract.

Particularly, when a vehicle V_e passes through the St_2 ETC station; It sends their data to the Ethereum blockchain for identification by comparing their produced unique identifier (i.e., UV_ID) to the list of registered vehicles. If the vehicle is verified, the smart contract retrieves the vehicle's current and previous locations. In our system we uses public-private key authentication to ensure that location information comes from a trusted source. The location data is signed with the vehicle owner's private key. Then, any user verifies the location using the owner's public key.

When the vehicle V_e reaches a distance present in the smart contract, the vehicle owner is charged a certain amount. In fact, The distance between the entry and exit toll is calculated using the euclidean distance formula shown in Eq. 5. Let $A(x_1, y_1)$ be the given starting point of V_e at the entry ETC station; and Let $B(x_2, y_2)$ represents the exit coordinates of V_e at the exit ETC station.

$$T_d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (5)$$

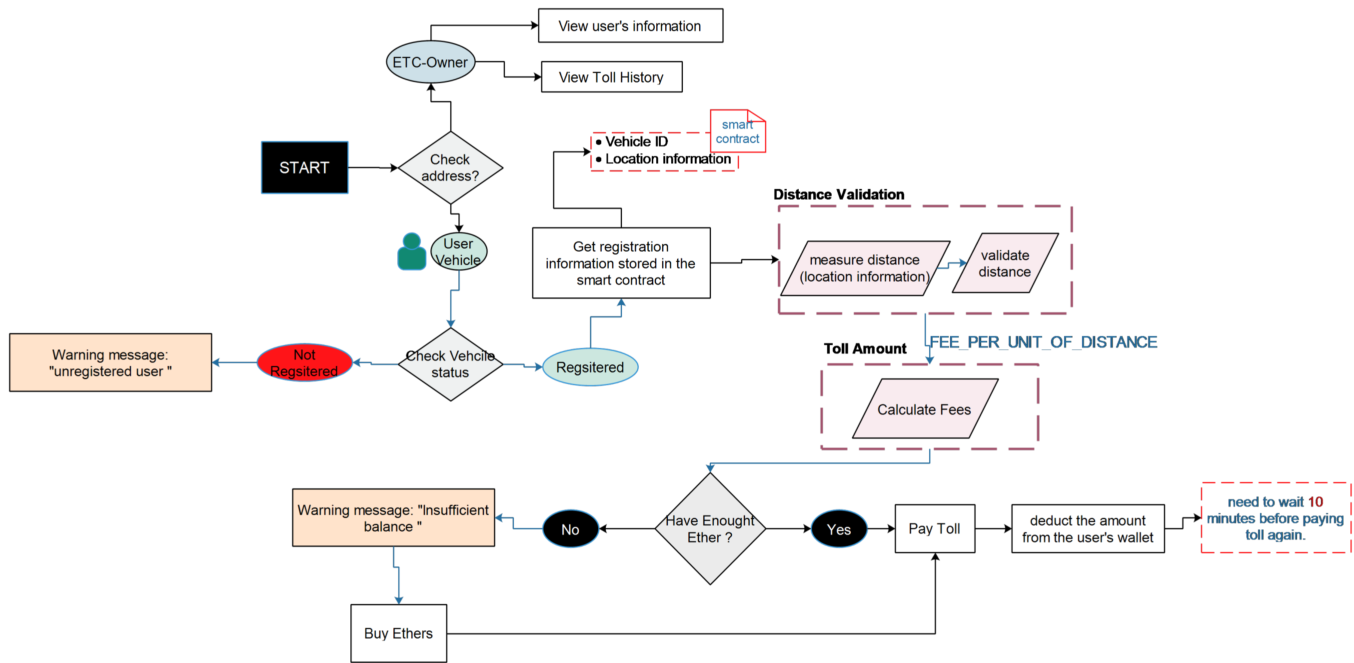

Fig. 7. Data collection and distance calculation methodology: ETC exit toll.

where T_d is the traversed distance between the entry and exit tolls on the highway; (x_1, y_1) and (x_1, y_2) respectively.

For example, let's say that a registered vehicle passes through the Entry ETC point located at $(2, 3)$ and the Exit ETC point located at $(6, 7)$. The smart contract can verify that the vehicle is registered and retrieve its entry location. Then, it can calculate the distance traveled by the vehicle using the following formula:

$$\begin{aligned} T_d &= \sqrt{(6-2)^2 + (7-3)^2} \\ &= \sqrt{16+16} \\ &= \sqrt{32} \\ &\approx 5.66 \end{aligned}$$

This means that the vehicle has traveled a distance of approximately 5.66 units between the Entry and Exit ETC points. Once the distance T_d has been calculated, the price is determined by multiplying T_d by the fee per unit of distance (denoted as f_d) which is expressed in terms of cryptocurrency units. For example, if the f_d is 0.01 Ether and the distance traveled by the vehicle is 5.66 kilometers, then the price is expressed as:

$$\begin{aligned} fees &= T_d \cdot \text{fee per unit of distance} \\ &= 5.66\,\text{km} \cdot 0.01\,\text{ETH/km} \\ &= 0.057\,\text{ETH} \end{aligned} \tag{6}$$

This means that the smart contract deducts approximately 0.057 ETH from the vehicle owner's account as the price for traveling between the Entry and Exit ETC points.

In this study, we employed a precision arithmetic library , provided by OpenZeppelin[†], called SafeMath[‡]. It is used in the smart contract to verify that the deducted fees are accurate. Particularly, SafeMath is used to prevent overflow/underflow. In this work, Safemath is used to perform the multiplication of the distance by the fee per unit of distance in order to ensure that the result is accurate and within the expected range.

Fig. 8 is a solidity code example that uses SafeMath library to perform the fee calculation. In this example, we first import the SafeMath library using the import statement. This library is applied to the uint256 data type (i.e., using statement).

We, then, define a constant FEE-PER-UNIT-OF-DISTANCE to represent the fee per unit of distance in the contract, and a function $calculateFees()$ that takes a distance parameter and returns the calculated fees. Inside the $calculateFees$ function, we use the SafeMath library's multiplication function $mul()$ to multiply the distance by the FEE-PER-UNIT-OF-DISTANCE, and assign the result to a variable fees. We then return the calculated fees.

By this way, we can ensure that the smart contracts handle arithmetic operations safely and avoid vulnerabilities related to integer overflow/underflow, which are common sources of security issues in blockchain applications. Meanwhile, we can ensure that the calculated fees are accurate and within the expected range.

D. Smart Contracts Security: Vulnerabilities

Smart contracts [16] are typically written in high-level programming languages such as Solidity. These high-level

[†]https://github.com/OpenZeppelin/openzeppelin-solidity
[‡]OpenZeppelin Solidity: SafeMath Library.
https://github.com/OpenZeppelin/openzeppelinsolidity/blob/master/contracts/math/SafeMath.sol

```
1  import "./SafeMath.sol";
2
3  contract MyContract {
4      using SafeMath for uint256;
5
6      uint256 constant FEE_PER_UNIT_OF_DISTANCE = 10; // 0.01 ETH/km
7
8      function calculateFees(uint256 distance) public view returns (uint256) {
9          uint256 fees = distance.mul(FEE_PER_UNIT_OF_DISTANCE);
10         return fees;
11     }
12 }
13
```

Fig. 8. Using SafeMath to perform an accurate fee calculation.

languages are then compiled into bytecode, which is a lower-level language that can be executed by the blockchain's virtual machine.

When a smart contract is executed, it is triggered by a blockchain transaction, which contains the necessary input data for the smart contract to perform its functions. The blockchain transaction is then processed by the blockchain network's nodes, which run the smart contract code on the virtual machine. The virtual machine serves as the execution environment for the smart contract, providing it with access to the blockchain's data storage and network resources. Once the smart contract has completed its execution, the results are recorded on the blockchain as a new transaction, which becomes a permanent part of the blockchain's immutable ledger.

Smart contracts are designed to be trustless and secure, but they can still be vulnerable to various security threats and vulnerabilities [17] throughout their lifecycle.

One of the primary security threats is a smart contract being hacked, resulting in theft or manipulation of funds. This can occur due to coding errors or vulnerabilities in the smart contract's design, as well as attacks on the underlying blockchain network.

A Reentrancy Attack [18] is a type of vulnerability in smart contracts, where an attacker can repeatedly enter and exit a contract function before the original transaction is completed, allowing them to execute malicious code and potentially steal funds. Here are the best practices that we have based on to prevent Reentrancy Attacks:

1) Checks-Effects-Interactions: The Checks-Effects-Interactions pattern (CEI) [19] is a best practice for designing smart contracts to ensure their security and reliability. The pattern recommends structuring the smart contract code in three distinct phases:

Checks: During this phase, the contract verifies whether the conditions for contract execution have been met. These checks may include ensuring that the sender has the necessary balance, that the contract is in the correct state, and that the smart contract has not previously been executed.
Effects: In this phase, the smart contract executes the requested function or updates the contract's state. The contract may, for example, transfer fees, update a balance, or change a record in the contract's storage.
Interactions: In this phase, the smart contract interacts with other contracts or external systems. For example, the contract may call a function in another smart contract, make an external API call, or transfer funds to an external wallet.

Let's consider an example of a smart contract that demonstrates the use of this pattern to provide protection against Reentrancy Attacks as shown in Fig. 9:

Assuming we have a vulnerable smart contract and an attacker who can exploit it. The attacker deploys a malicious contract that calls the vulnerable contract's function $withdraw()$ repeatedly before the balance is updated, thus causing the contract to pay out more than what the user has in their account.

Let's consider the following vulnerable withdraw function in a smart contract: The withdraw function [see Fig. 9(A)] allows a user to withdraw a specified amount of Ether from their balance. The function first checks if the user has enough funds in their balance to withdraw the specified amount. Then, it uses the call function to transfer the specified amount of Ether to the user's address, and finally deducts the amount from the user's balance.

This code is vulnerable to a Reentrancy Attack because the external call function $msg.sender.callvalue : amount("")$ can execute arbitrary code, including calling the withdraw function again before the balance is updated. An attacker could repeatedly call the withdraw function, draining the contract's balance and leaving the user with an incorrect balance.

Let's assume that the contract has a starting balance of 100 ETH and the attacker is able to drain the contract's balance at a rate of 1 ETH per second. In this scenario, the attacker can drain the entire contract's balance in 100 seconds.

With CEI, the protected contract checks the user's balance, updates the balance, and then sends ether to the user's address. This ensures that the balance is updated before any external calls are made, making the contract safe from Reentrancy Attacks. Assuming the same starting balance of 100 ETH, the protected contract can resist the Reentrancy Attack because the attacker will not be able to drain the contract's balance before the user's balance is updated.

In the updated code (see Fig. 9), we have added a locked mapping to keep track of whether a user is currently withdrawing funds to prevent Reentrancy Attacks. The locked mapping is set to $true$ when the user starts withdrawing funds and set back to $false$ when the withdrawal is complete.

Suppose the user has initially 100 ETH in their balance, and they want to withdraw 50ETH. Here is how the updated withdraw function using the CEI pattern works:

1) The function checks that the user has a balance of at least 50 ETH and that the user is not currently withdrawing funds ($!locked[msg.sender]$).
2) The function sets the locked flag to $true$ for the user to indicate that they are currently withdrawing funds.
3) The function deducts 50 ETH from the user's balance.
4) The function executes the call function to transfer 50 ETH to the user's address.
5) The function checks that the call function was successful [require(success)]
6) The function sets the locked flag back to false for the user to indicate that the withdrawal is complete.

By updating the user's balance and setting the locked flag before executing the call function, we ensure that the user's balance is updated before any external interactions occur. This makes it impossible for an attacker to repeatedly call the withdraw function before the balance is updated, effectively preventing Reentrancy Attacks. In this way, the CEI pattern can protect smart contracts against Reentrancy Attacks and ensure their security.

By separating the checks, effects, and interactions into distinct logical components, this can ensure that our smart contract code is more secure and less likely to be vulnerable to attacks and helps to improve the clarity, maintainability, and modularity of the code.

2) Limiting GAS fees: A Reentrancy Attack occurs when a malicious user exploits a vulnerability in a smart contract to repeatedly call a function within that contract before the previous call has finished executing. This can lead to unintended behavior, such as the attacker being able to drain funds from the contract. One way to mitigate the risk of Reentrancy Attacks is to limit the gas fees for transactions that interact with smart contracts. This can be done by setting a maximum gas limit for these transactions, which would prevent the attacker from executing an excessive number of function calls within a single transaction.

Here is an example of how limiting gas fees can help to prevent a Reentrancy Attack: Considering the two codes depicted in Fig. 10. In the first code example [Fig. 10 (vulnerable code)], the withdraw function allows a user to withdraw a specified amount from their account balance. The function first checks that the user has enough funds, then transfers the funds to the user's address using the call function. However, there is a vulnerability in this contract that allows a malicious user to exploit the call function to repeatedly call the withdraw function before the previous call has completed. This can lead to the user receiving funds multiple times, effectively draining the contract balance.

To prevent this attack, we can limit the amount of gas that can be used by the withdraw function. We can do this by setting a gas limit using the gas keyword as shown in Fig. 10(updated code). we have set a gas limit of $100,000$ gas for the call function. This means that the withdraw function can only use a maximum of $100,000$ gas for each call. If an attacker tries to repeatedly call the withdraw function using more gas than the limit, the transaction will fail and any changes made by the function will be reverted. This helps prevent the Reentrancy Attack and protects the contract balance.

However, it is important to note that limiting gas fees alone may not be sufficient to prevent Reentrancy Attacks. It is also necessary to carefully audit smart contracts for vulnerabilities and to implement appropriate security measures, such as using the "withdrawal pattern" to prevent Reentrancy Attacks.

3) Withdrawal pattern: Here we demonstrate the importance of using withdrawal Pattern to protect the smart contract against reentrency attacks. As shown in Fig. 11(1), this is a sample contract vulnerable to reentracy attack. In this code it is clear that when using $msg.sender.call.value$ to transfer fees, this makes it susceptible to Reentrancy Attacks. Thus, an attacker may create a hacker contract and repeatedly call the withdraw function to to drain the contract's balance before their own balance is updated.

To effectively preventing re-entrancy attacks, a secure version of the simple example [Fig. 11(1)] as shown in Fig. 11(2), where we define a $requestWithdrawal()$ method. We set the $withdrawalAllowed = true$. Then, the withdraw function checks if withdrawal is allowed. Meanwhile, it ensures that there are sufficient funds and disables further withdrawals until the current one is completed. This prevents the continuous calls of function until the the contract's funds are drained. After these checks, the function performs the transfer using $payable(msg.sender).callvalue : amount("")$, and the sender's balance is updated.

V. System Design and Performance Evaluation

This section presents the design of the blockchain-based system for electronic toll collection. Particularly, it highlights the technical aspects of the implementation and describes the data security measures to evaluate the effectiveness of the blockchain-based ETC system in enhancing data security and accuracy of fees.

A. System Design

In this subsection, we present the blockchain ecosystem (shown in Fig. 12) used to analyze and evaluate our proposal. The implementation mainly include: User Interface and Ethereum blockchain which includes smart contract, Ganache, metamask and truffle framework.

Particularly, in this work, we make a decentralized application (Dapp) in which users may access through web browsers. To do that, we use the truffle development framework to develop, test and deploy the smart contract. We choose to run an Ethereum node locally and use the Ganache tool which allow us to connect and distribute the EVM workload across the nodes in the blockchain network. Ganache offers by default 10 dummy account addresses and private keys (i.e. one per each account). Thus, in spite of connecting to the entire network, we basically connect to the local Ethereum

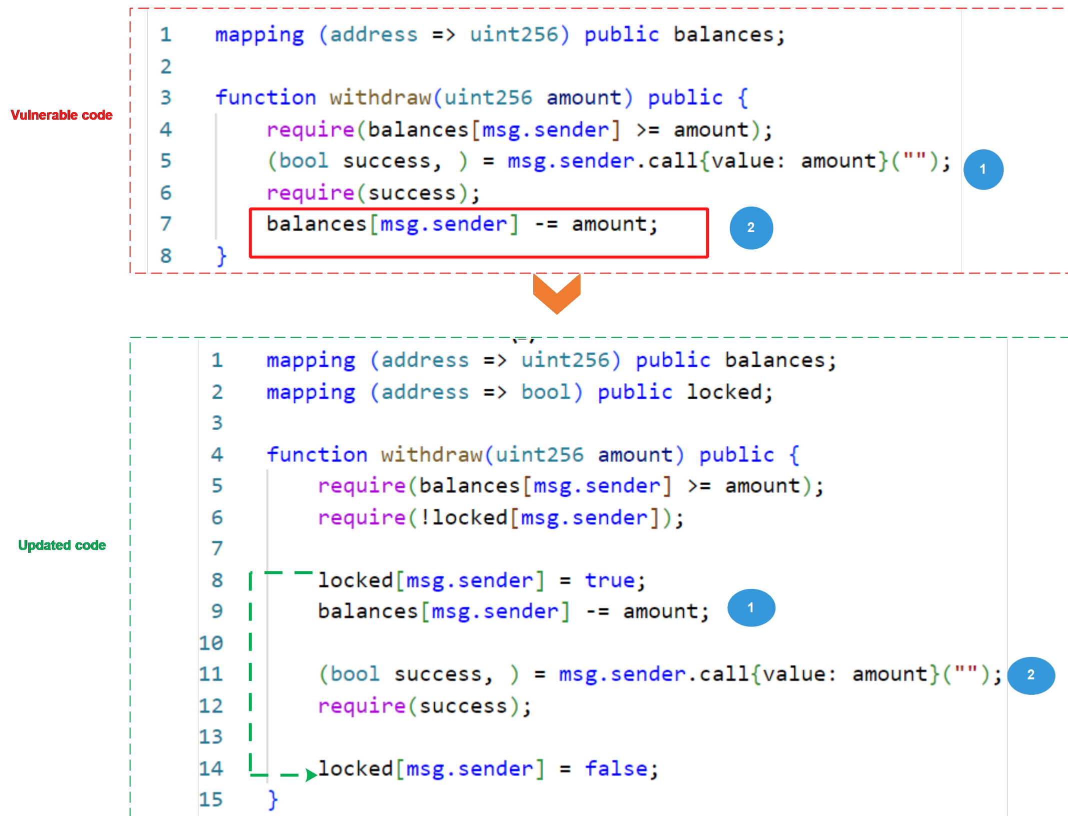

Fig. 9. CEI pattern: (A) Vulnerable code and (B) Updated code.

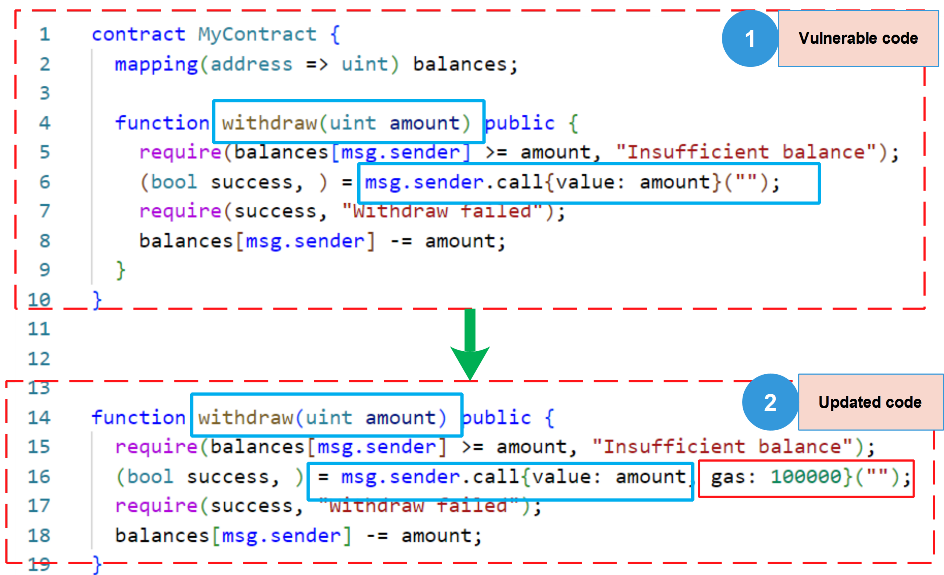

Fig. 10. (1) Vulnerable code and (2) Updated code (Limiting GAS fees).

network using Ganache. So, the entire Ethereum network is located locally in the computer. Each Ganache account address is considered as an EOA and will be assigned to one vehicle user. Particularly, we consider two types of EOA accounts: user account and owner account. We are considering an accounts table that manage the whole accounts addresses and we assume that $owner_{addr} = Accounts[0]$. This owner address $owner_{addr}$) is the one which deploy the smart contract in the blockchain. The deployment of the smart contract to the Ethereum Blockchain creates a contract address. This contract Address needs to be replaced after each deployment.

```
1  contract VulnerableContract {
2      mapping(address => uint256) public balances;
3
4      function withdraw(uint256 amount) public {
5          require(balances[msg.sender] >= amount, "Insufficient balance");
6
7          // Vulnerable to reentrancy attack
8          (bool success, ) = msg.sender.call{value: amount}("");
9          require(success, "Transfer failed");
10
11         balances[msg.sender] -= amount;
12     }
13 }
```

```
1  contract SecureContract {
2      mapping(address => uint256) public balances;
3      mapping(address => bool) public allowedWithdrawals;
4
5      function deposit() public payable {
6          balances[msg.sender] += msg.value;
7      }
8
9      function requestWithdrawal(uint256 amount) public {
10         require(balances[msg.sender] >= amount, "Insufficient balance");
11         allowedWithdrawals[msg.sender] = true;
12     }
13
14     function withdraw(uint256 amount) public {
15         require(allowedWithdrawals[msg.sender], "Withdrawal not allowed");
16         require(balances[msg.sender] >= amount, "Insufficient balance");
17
18         allowedWithdrawals[msg.sender] = false;
19
20         (bool success, ) = payable(msg.sender).call{value: amount}("");
21         require(success, "Transfer failed");
22
23         balances[msg.sender] -= amount;
24     }
25 }
```

Fig. 11. (1) Vulnerable code and (2) Updated code (withdrawal pattern).

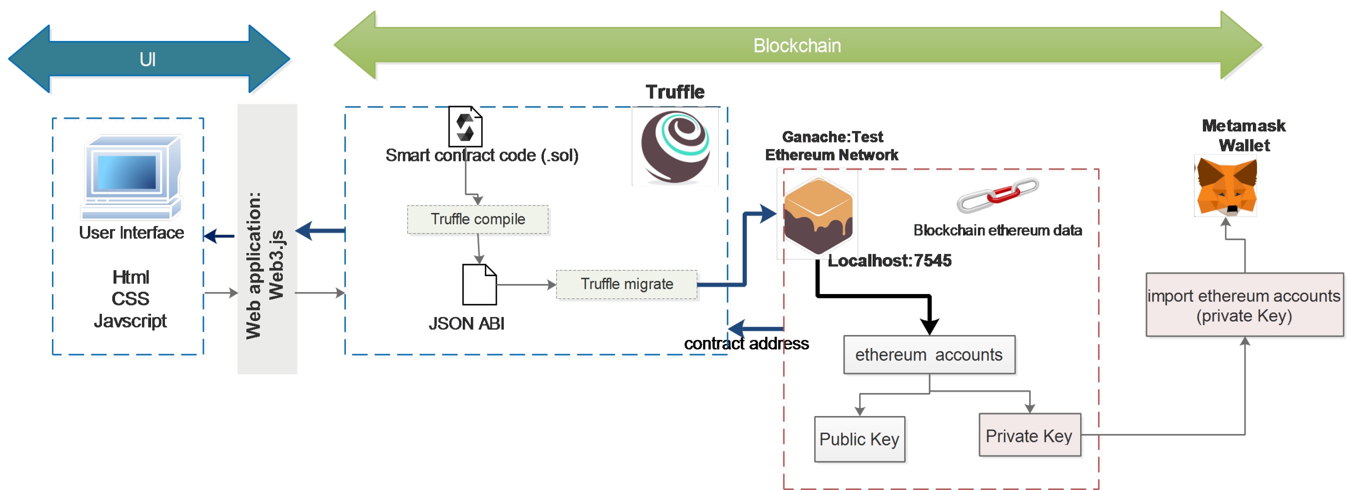

Fig. 12. User interface and blockchain framework.

TABLE III. ABBREVIATIONS AND SYMBOLS

Prerequisites	Version
NodeJs	v16.17.1
Truffle	v5.4.29
Ganache	V2.5.4
Web3.js	v1.5.3

This blockchain framework is accessed on a web browser like google chrome using Metamask. In fact, we have created a frond-end (user interface) with Node.js Web server, HTML and CSS. In order to interact with the distributed application, we use Metamask application which allows users to run Dapp directly in the browser without running a full Ethereum of node. The main prerequisites implementation tools are listed in Table III.

In Fig. 13, we depict a flowchart that provides a high-level overview of the process for using Truffle with MetaMask and acquiring Ether to develop, test, and deploy smart contracts. In fact, In step (1), we first use the *init* command to initialize a new project with the default contracts, migrations and tests folders and $truffle–config.js$ file [as shown in step (2)] that represent the basic template to start new project. In Step (3) we configure the $truffle–config.js$ which contains the network endpoints for deployments. For example, we have used the following lines for Ganache deployment running on $localhost$: 8545 for migration and testing.

```
module.exports = {
  networks: {
    development: {
      host: '127.0.0.1',
      port: 8545,
      network_id: '*'
    }
  }
}
```

Migration scripts are used for deploying smart contracts to the Ethereum network. After a successful migration process(i.e., using the command `truffle migrate --reset`) in the network object creates new migration scripts as shown in steps (6) and (7). After that, as shown in steps (8) and (9), the deployment of the smart contract require the check of the wallet. If the wallet has has enough ether to cover the

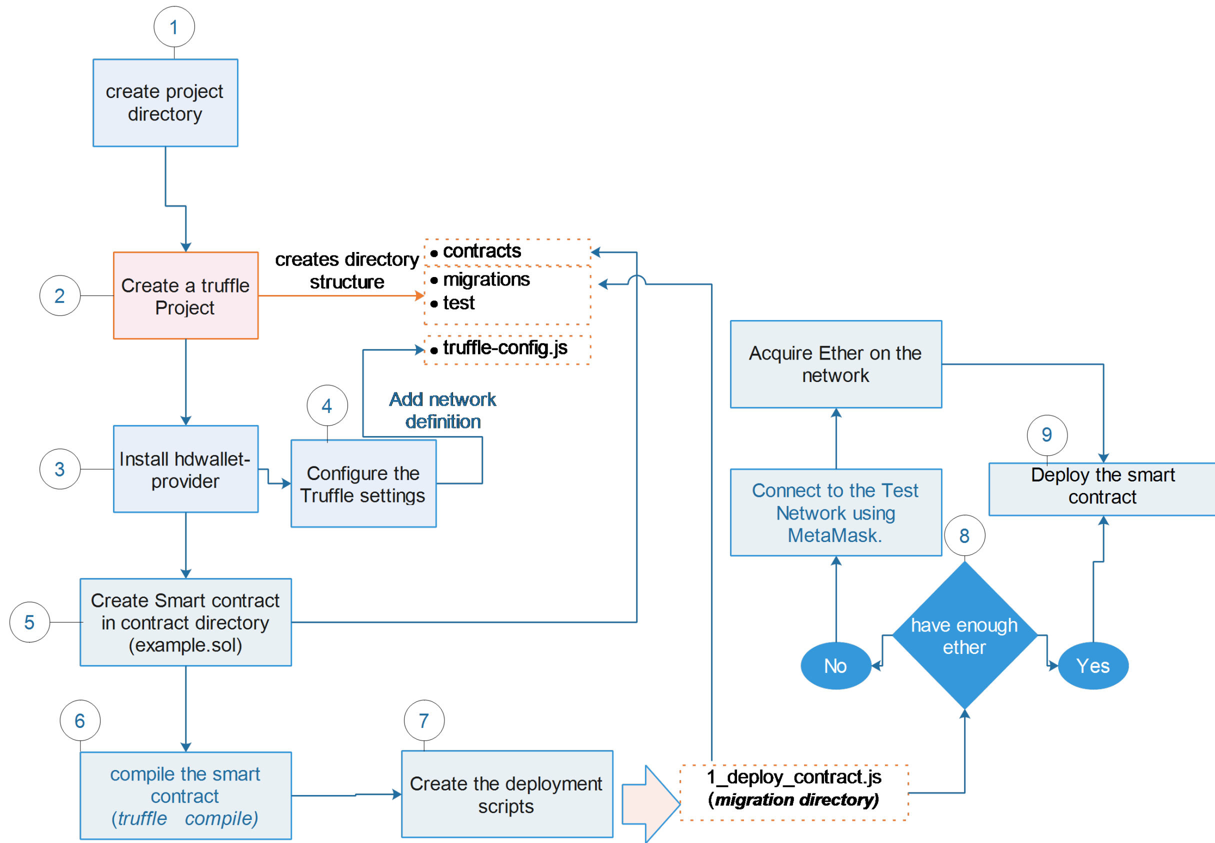

Fig. 13. Smart contract deployment using truffle.

TABLE IV. TRANSACTION STATISTICS

NB transactions	100	200	300	400	500	600	700	800	900	1000
% Valid Tr	99.0	99.0	99.33	99.5	99.4	99.5	99.57	99.5	99.56	99.5
% Non-Valid Tr	1.0	1.0	0.67	0.5	0.6	0.5	0.43	0.5	0.44	0.5
Accuracy-fees	99.99	99.98	99.97	99.97	99.97	99.91	99.92	99.84	99.84	99.83

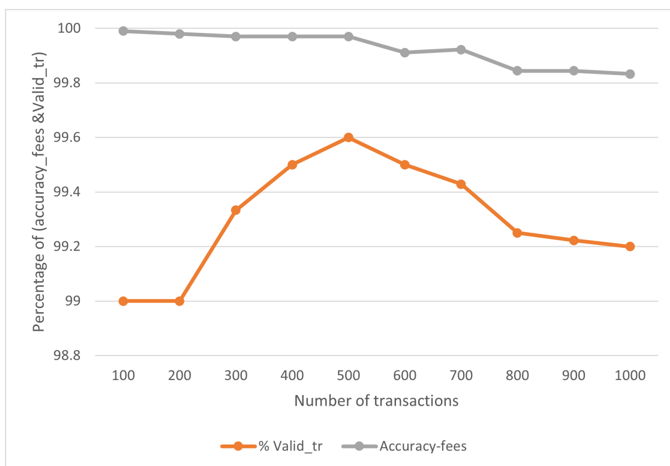

Fig. 14. Accuracy-fees and valid-transactions vs number of transactions.

deployment cost and any transactions, the smart contract is deployed on the network. If not Ethers must be acquired on the network.

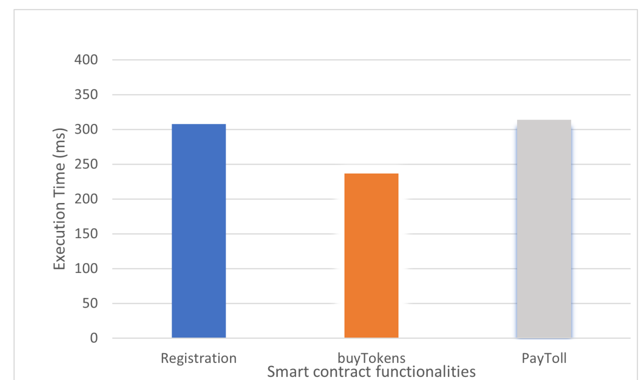

Fig. 15. Execution time.

B. Performance Evaluation

In this subsection, we evaluate the efficiency of the smart contract regarding the Reentrency Attacks and accuracy of fees calculation. For that purpose, two metrics are considered (i.e., Accuracy of fees calculation and execution time).

Let T be a set of n transactions denoted as $T = t_1, t_2, ..., t_n$, where each transaction t_i has an expected distance and toll fees denoted as di_{exp} and fi_{exp} respectively.

For each transaction t_i in T, the actual toll fees fi_{act} is calculated based on the distance di_{act} and the toll rate r which represents the fee per unit of distance. This formulate is expressed in Eq.7.

$$fi_{act} = di_{act} \times r \tag{7}$$

The accuracy of the ETC fees calculation, denoted as $ti^{fees}_{accuracy}$, for a single transaction t_i is then given by:

$$\begin{aligned} ti^{fees}_{\text{accuracy}} &= \left[1 - \left(\frac{|fi_{exp} - fi_{act}|}{fi_{exp}}\right)\right] \times 100\% && (8) \\ &= \left[1 - \left(\frac{|di_{exp} \times r - di_{act} \times r|}{di_{exp} \times r}\right)\right] \times 100\% && (9) \\ &= \left[1 - \left(\frac{|r(di_{exp} - di_{act})|}{di_{exp} \times r}\right)\right] \times 100\% && (10) \\ &= \left[1 - \left(\frac{|di_{exp} - di_{act}|}{di_{exp}}\right)\right] \times 100\% && (11) \end{aligned}$$

The average accuracy for the set of transactions T is expressed by:

$$\begin{aligned} T_{accuracy} &= \frac{1}{n}\sum_{i=1}^{n} ti^{fees}_{accuracy} && (12) \\ &= \frac{1}{n}\sum_{i=1}^{n}\left[1 - \left(\frac{|di_{exp} - di_{act}|}{di_{exp}}\right)\right] \times 100\% && (13) \end{aligned}$$

where n represents the total number of transactions.

In this work, let's assume we have a secure smart contract that accurately calculates the fees for a given distance. We will discuss the accuracy of our smart contract in the presence of a hacker smart contract (hacker node) that tries to modify the distance values. We calculate the average accuracy of fees using Eq. 13 for five transaction sets denoted by: $n = 100$, $n = 200$, $n = 300$, $n = 400$, and $n = 500$.

This will help us understand how the accuracy of fees is affected as the number of transactions increases.

Fig. 14 shows that as the number of transactions increases, the average accuracy of fee calculation decreases, indicating a slight impact of the hacker smart contract on our secure smart contract.

This result shows that, while the hacker smart contract did have an impact on the accuracy of fees calculation, the overall impact on the entire set of transactions is relatively small. In fact, the high percentage of fees accuracy reflect that the proposed decentralized system is highly accurate.

In the same figure (i.e. Fig. 14), we observe the percentages of valid transactions. We remark that the number of transactions increases from 100 to 1000, the percentage of valid transactions remains consistently high (ranging from 99% to 99.5%) as shown in Table IV. This indicates that with a larger number of transaction, the hacker smart contract may affect a few specific transactions. Particularly, this indicates that the smart contract's security patterns are resilient even when dealing with a larger number of transactions.

For example if we take these two cases from the Table IV: (i) 500 transactions resulting in 0.6% of non-valid transactions and 99.97% of fees accuracy; ii) 200 transactions resulting in 1% of non-valid transactions and 99.98% of fees accuracy. Despite the relatively higher percentage of non-valid transactions at 1% in the second case, the fees accuracy remained high. This result is explained by the fact that hacker smart contract had a minimal impact on the average accuracy of fees, signifying that the difference between the expected and actual average of fees calculation was extremely low.

The performance of the smart contract is evaluated based on the execution time metric. Fig. 15 shows the execution time of the three basic functions of the given smart contract (Registration, Buy_toll, Pay_Toll) that were called by a specific transaction. We can see that the registration and the pay_toll take $307ms$ and $314ms$, respectively. The Buytoll function only takes about $236ms$ less than registration and the pay_toll functions. The functions (registration and the pay_toll) need more execution time, which is reasonable, because they include complex calculations such as verification of vehicle's status and distance validation at the exit toll.

VI. Conclusion

In conclusion, this paper tackles the vulnerabilities of current Electronic Toll Collection (ETC) systems, including privacy issues and potential attacks such as the Reentrancy Attack. To address these challenges, we proposed an innovative solution leveraging Ethereum blockchain and smart contracts for automated payments within the Internet of Vehicles (IOV) framework. Our main goals are to authenticate vehicle data, automatically deduct toll fees from users' wallets, and safeguard against smart contract Reentrancy Attacks while protecting sensitive distance-related information.

Specifically, we introduced an end-to-end verification algorithm that functions at both entry and exit toll points, providing a robust solution to these issues. We evaluated the system's performance on a private blockchain, and the results show that this decentralized approach not only enhances security but also ensures accurate payment processing.

In future research, we plan to integrate deep learning algorithms to further enhance the system's capabilities. By incorporating deep learning, we aim to detect anomalies and potential fraud in real-time, improving the overall reliability and security of the ETC system. This addition will provide an even more comprehensive and intelligent solution for managing toll payments in the context of the Internet of Vehicles.

Acknowledgments

The authors gratefully acknowledge the approval and the support of this research study by the grant no. SCIA-2023-12-2233 from the Deanship of Scientific Research at Northern Border University, Arar, KSA.

REFERENCES

[1] J. Wang, R. Zhu, T. Li, F. Gao, Q. Wang and Q. Xiao, *ETC-Oriented Efficient and Secure Blockchain: Credit-Based Mechanism and Evidence Framework for Vehicle Management,* in IEEE Transactions on Vehicular Technology, vol. 70, no. 11, pp. 11324-11337, Nov. 2021, doi: 10.1109/TVT.2021.3116237

[2] Das, D., Banerjee, S., Biswas, U. *Design of a secure blockchain-based toll-tax collection system.* In: Micro-electronics and telecommunication engineering, pp. 183–191. Springer, Singapore (2022).

[3] Shanmukha Makani, Rachitha Pittala, Eitaa Alsayed, Moayad Aloqaily and Yaser Jararweh, *A survey of blockchain applications in sustainable and smart cities*, Journal: Cluster Computing, 2022

[4] Sahoo, Sujit Sangram, Aravind R. Menon, and Vijay K. Chaurasiya. *Secure Blockchain Model for Vehicles toll Collection by GPS tracking: A case study of India.* 2022 IEEE India Council International Subsections Conference (INDISCON). IEEE, 2022.

[5] Banerjee S, Das D, Biswas M, Biswas U (2020), *Study and survey on blockchain privacy and security issues.* In: Williams I (ed) Cross industry use of blockchain technology and opportunities for the future. IGI Global, pp 80–102. https://doi.org/10.4018/978-1-7998-3632-2.ch005

[6] Guo, Yihao, et al. *Vehicloak: A Blockchain-Enabled Privacy-Preserving Payment Scheme for Location-Based Vehicular Services.* IEEE Transactions on Mobile Computing (2022).

[7] Deng, Xinyang, Gao, Tianhan. (2020). *Electronic Payment Schemes Based on Blockchain in VANETs*. IEEE Access. PP. 1-1. 10.1109 ACCESS.2020.2974964.

[8] Chiu, Wei-Yang, and Weizhi Meng. *EdgeTC-a PBFT blockchain-based ETC scheme for smart cities.* Peer-to-Peer Networking and Applications 14 (2021): 2874-2886.

[9] Buterin, V. *Ethereum: a next generation smart contract and decentralized application platform.* https://github.com/ethereum/wiki/wiki/White-Paper (2013)

[10] Clack, C.D., Bakshi, V.A., Braine, L. *Smart contract templates: foundations, design landscape and research directions.* CoRR abs/1608.00771 (2016)

[11] Szabo, N. *Formalizing and securing relationships on public networks. First Monday 2(9) (1997),* http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/ fm/article/view/548

[12] Ikram Ali, Alzubair Hassan, and Fagen Li, *Authentication and privacy schemes for vehicular ad hoc networks (VANETs): A survey*, Vehicular Communications Volume 16, April 2019, pp. 45-61, doi: 10.1016/j.vehcom.2019.02.002

[13] R. Jabbar et al., *Blockchain Technology for Intelligent Transportation Systems: A Systematic Literature Review,* in IEEE Access, vol. 10, pp. 20995-21031, 2022, doi: 10.1109/ACCESS.2022.3149958

[14] Liu and Cao. 2018. *Reguard: finding reentrancy bugs in smart contracts.* In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. ACM, 65–68

[15] Wood, G. *Ethereum: a secure decentralised generalised transaction ledger.* gavwood.com/paper.pdf (2014)

[16] Zou, W.; Lo, D.; Kochhar, P.S.; Le, X.-B.D.; Xia, X.; Feng, Y.; Chen, Z.; Xu, B. *Smart Contract Development: Challenges and Opportunities.* IEEE Trans. Softw. Eng. 2021, 47, 2084–2106.

[17] Qian, P.; Liu, Z.; He, Q.; Zimmermann, R.; Wang, X. Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models. IEEE Access 2020, 8, 19685–19695.

[18] Mehar, M.I.; Shier, C.L.; Giambattista, A.; Gong, E.; Fletcher, G.; Sanayhie, R.; Kim, H.M.; Laskowski, M. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. J. Cases Inf. Technol. 2019, 21, 19–32

[19] Maximilian Wohrer and Uwe Zdun. Smart contracts: security patterns in the Ethereum ecosystem and solidity. In 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pages 2–8. IEEE, 2018.