KU LEUVEN

Applying Smart Contracts to Secure Car Sharing Systems

Akash Madhusudan

Thesis submitted for the degree of Master of Science in Artificial Intelligence, option Big Data Analytics

Thesis supervisor:
Prof. dr. ir. Bart Preneel

Assessors:
Prof. dr. ir. Frederik Vercauteren
Andreas Put

Mentors:
Dr. Mustafa A. Mustafa
Dr. Iraklis Symeonidis

Academic year 2017 – 2018

Preface

I heard about the blockchain technology in a lot of social gatherings with my fellow IT friends where they tried to explain it to me, but it was too complicated. I only got a chance to really learn something about it when I started this master thesis. I did extensive research and I was astonished by its applications. This technology is an asset with capabilities of offering a huge impact in several sectors, hence making it the perfect subject for my masters thesis. A lot of people supported me with their valuable inputs and suggestions. First of all I would like to thank my mentors Iraklis Symeonidis and Mustafa A. Mustafa and promotor Bart Preneel for showering me with their knowledgeable suggestions and pushing me whenever I was stuck. Iraklis and Mustafa even took out their personal time from vacations to answer me everytime I emailed them with questions. I would also like to thank Ren Zhang for helping me understand the technical aspects of blockchain and my friends Femke, Kevin, Arno, Cansu, Danica, Ishaan along with my sister Akanksha for supporting me in anything and everything after a long day of working on my thesis. There have been countless occassions when you guys have uplifted my mood after a long day of hard work. Finally, I would like to thank my mother, without whom my academic journey would not be a possibility.

Akash Madhusudan

Contents

Abstract

The appearance of Consumer-to-Consumer (C2C) markets has introduced a possibility for individuals to share their personal underutilised assets for a short- or long-term. However, their reliance on a trusted third party leads to various privacy concerns. Car sharing systems such as car2go or Zipcar are gaining popularity, and the use of in-vehicle telematics has introduced systems that allow users to share their cars conveniently without requiring traditional keys. These systems are called key-less car sharing systems. Although such systems offer convenience to users, they also introduce several security and privacy challenges.

A solution that addresses some of these challenges and allows individuals to share their personal car in a secure and privacy-friendly way is SePCAR, a protocol for car access provision proposed by Symeonidis et al. [48]. This thesis is about designing a decentralised booking and payments platform for SePCAR so as to eradicate these security and privacy challenges and allow these car sharing systems to operate without a trusted intermediary. Our implementation makes the use of smart contracts deployed on the Ethereum blockchain that provides full-fledged car sharing functionalities along with various countermeasures to tackle malicious behaviour.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations

B2C	Business-to-Consumer
C2C	Consumer-to-Consumer
DDOS	Distributed Denial of Service
DHIES	Diffie-Hellman Integrated Encryption Scheme
ECC	Elliptic Curve Cryptography
ECIES	Elliptic Curve Integrated Encryption Scheme
ENC	Encryption
EVM	Ethereum Virtual Machine
KA	Key Agreement
KDF	Key Derivation Function
MAC	Message Authentication Code
MPC	Multi-Party Computation
PKI	Public Key Infrastructure
PoE	Proof of Existense
P2P	Peer-to-Peer
RSA	Rivest–Shamir–Adleman
TTP	Trusted Third Party

Chapter 1

Introduction

Since its emergence, the e-commerce business has been mainly monopolised by Business-to-Consumer (B2C) platforms. However, the appearance of Consumer-to-Consumer (C2C) platforms has evolved this simple sale and resale market to an advanced version that enables short- and long-term rentals of various assets. Using this market, people come together to share their personal underutilised assets for financial gain while also helping others who are in temporary need of these assets. Statistics reflect the popularity of car sharing services by showing a hike in their usage from 2012 to 2014 by 170 % (4.94 million) [18].

All these platforms rely on a Trusted Third Party (TTP) that poses many disadvantages; customers have to give away their personal information when signing up which leads to various privacy concerns as their private information can be easily misused. An instance that highlights this misuse is the tool called 'Hell' which was used by Uber to spy on rival company drivers [29]. In fact, in 2017 their mobile app was also proven to collect the location of their users *round-the-clock* [26], which is a clear privacy violation. Moreover, these TTP's are vulnerable to Distributed Denial of Service (DDoS) attacks and their failure shuts down the whole system making them a single point of failure. Hence, it can be deduced that this Peer-to-Peer (P2P) economy is not a true flavour of Peer-to-Peer networks as they lack full decentralisation due to their reliance on a centralised intermediary [46].

A way to enable full decentralisation in this P2P economy are 'Smart Contracts', a term coined by Nick Szabo [50] in 1997. These smart contracts enforce a set of predefined rules which are coded as logic to orchestrate agreement between different entities. Szabo hypothesised a digital security system for cars using these smart contracts but due to the lack of a digital infrastructure this system was not achievable. However, with the emergence of cryptocurrencies like Bitcoin [39] and Ethereum [10], Szabo's vision has become achievable.

Despite being the most valued and popular cryptocurrency in the world, Bitcoin cannot extend its functionalities to complex contracts due to the limitations in its

scripting language. Ethereum addresses these limitations by supporting several scripting languages to write such smart contracts.
Smart contracts can be used to allow people to build a community where they can rent their personal assets in a secure and privacy-friendly way thus evolving the e-commerce business to a whole new decentralised level.

Currently, a rental contract between two people requires a mediation between the requirements of the owner and the customer using a platform which describes the rental asset and its fee enabling the customer to rent it if they agree with the terms. Payment of this fee usually requires a monetary asset issued by a bank. In the case of car sharing systems, there are a lot of applications that enable the rental process but there is a severe lack of a community where a person can rent his personal car in a secure and privacy-friendly way.

Symeonidis et al. [48] proposed a car sharing protocol, namely SePCAR, that offers a guarantee on the security of the car, privacy of the customer and provides countermeasures in case a customer is denied access to the car. Along with these guarantees, this protocol also discloses the identities of the owner, the customer and the car in case of an incident which involves these entities. It also provides access provision even while the car is not able to connect to a network, and enables key revocation whenever it might be needed, for instance if the customer's mobile device gets stolen. Multiparty computation is the core of this protocol, that allows various individual parties to collectively work and execute functions while keeping their respective share of information private [17].

This work is focused on implementing a secure and privacy-friendly booking and payments system using smart contracts which can be incorporated in SePCAR. We provide a secure and privacy-friendly smart contract deployed on the Ethereum blockchain that allows a user to share their personal car and is capable of carrying out the entire car rental process ranging from booking to payments and fraud prevention all while being mostly decentralised. We have made efforts to bridge the gap between our smart contract and SePCAR by fulfilling most of its core security and privacy requirements and evaluated its performance in terms of the requirements met and costs incurred by deploying and executing these contracts.

The remainder of the thesis is organised as follows, Chapter 2 provides the necessary background on car sharing systems, blockchain, cryptocurrency, smart contracts and ECIES (the public-key cryptography used in our implementation) and discusses the related work done in these fields. Chapter 3 gives an overview of our smart contract, lists the design requirements (functional, security and privacy) and describes the methodology we followed in detail by listing various case studies. This chapter also evaluates our smart contract on two grounds, namely, the requirements met and the total cost incurred to the user and lists the results. Finally, Chapter 4 concludes the thesis while listing some lessons learnt, directions for future work and also lists some limitations of our implementation.

Chapter 2

Background & Related Work

Our thesis focuses on different areas of technology, namely blockchain, cryptocurrency, smart contracts and cryptography technologies. This chapter aims to provide a detailed insight into what these technologies actually are and how they work, describing also some existing work in these fields. With this detailed insight, the reader will have a better understanding of our implementation that we provide in Chapter 3.

2.1 Car Sharing Systems

Car sharing systems can be described as models that allow people to share their personal car for commercial or private use by others. It includes many applications ranging from carpooling to private rentals. This section is divided in two parts, the first part lists the existing work on car sharing systems and in the second part we elaborate on the work proposed by Symeonidis et al. [48].

2.1.1 Existing Work

A privacy-preserving protocol in which the fee owed by the consumer of a shared car is calculated using location data has been designed by Troncoso et al. [54]. A similar privacy-preserving system to calculate an annual toll is proposed by Balasch et al. [5]. When it comes to eco-friendly systems, Mustafa et al. [38] proposes an anonymous electric vehicle charging protocol, Lee et al. [34] proposes an electric car sharing system which enables users to use their smart phones to book available electric cars and Cepolina and Farina [12] propose a transportation system for urban areas that uses a fleet of electric autonomous vehicles that users can book for one-way trips. However, there is a lack of systems that enable secure and private car sharing using mobile devices. A system which uses two-factor authentication, for instance smart cards, to offer security is proposed by Dmitrienko and Plappert [19] but it does not consider the system user's privacy. A protocol that satisfies the desired security and privacy properties of a car sharing system is proposed by Symeonidis et al. [48], namely SePCAR, based on their earlier work [49]. It enables users to make bookings

and retrieve the access token in a secure and privacy-friendly way, and this process of access token creation is the main difference of SePCAR and the system proposed by Dmitrienko and Plappert [19]; their system assumes the user already has an access token and discusses a secure way to open the car using that access token.

2.1.2 System Model

In this part we will give a high-level description of SePCAR along with its functional components, and we will also list its security and privacy requirements.

2.1.2.1 Functional Components

Figure 2.1 consists of the functional components of SePCAR namely the Owner, Consumer, Portable Device (PD), Key-less Sharing Management Server (KSMS), Car Manufacturer (CM), Public Ledger (PL), On-Board Unit (OBU) and Authorities, and are explained as follows:

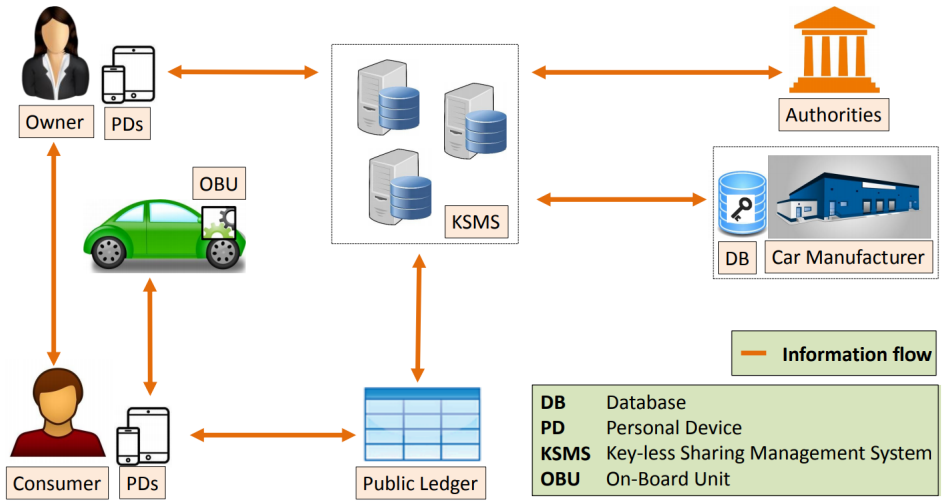

Figure 2.1: A Physical Key-less Car Sharing System. [48]

- Owner – a user or a group of users willing to share their car.
- Consumer – the users who want to rent a car.
- Portable Device (PD) – a device that contains a Key-less Sharing Application (KSApp) which enables users to communicate with each other and the KSMS.
- Key-less Sharing Management Server (KSMS) – the party responsible for the creation of an access token using multi-party computation while having no

access to the car key and booking details. In multi-party computation, several parties collectively compute a function while keeping their share of inputs hidden from each other.

- Car Manufacturer (CM) – the entity that produces cars and also has a database with different car keys and communicates regularly with KSMS.
- Public Ledger (PL) – an entity that stores temporary access tokens generated by KSMS where consumers are allowed to retrieve and use these tokens.
- On-Board Unit (OBU) – an embedded component in the car used for communication with the consumers using NFC or bluetooth.
- Authorities – entities responsible for ensuring that the entire system is legal as well as for resolving any disputes between users.

KSMS, CM and PL are considered to be honest but at the same time meddlesome; they do not have malicious intentions but are interested in gaining private information of the users. The OBU is assumed to be secure and trustworthy while the PD's are assumed to be the opposite, unsecured and untrustworthy because of their high susceptibility to malicious attacks [48].

2.1.2.2 High Level Description

In SePCAR [48], the first step involves the mutual agreement of the booking details by both the owner and the consumer. These details hold sensitive information such as the identities of the owner, the consumer and the car along with the location and the usage duration of the car. These mutually agreed booking details are then sent to the KSMS where they are encrypted and an access token is generated. This access token is then used by the consumer to access the car without revealing any of their private information. SePCAR consists of the forensics property that ensures that in case of an incident, the privacy of the consumer may be lifted and revealed to the authorities; it is obtained by a collective effort of the independent parties in the Multi-Party Computation (MPC) protocol. Moreover, the MPC is also used to guarantee confidentiality of car key and booking details by dividing the whole process in two stages, Car Key Retrieval and Access Token Generation phase.

Car Key Retrieval Phase: The information provided by the owner is utilised by the KSMS server to generate an access token for the car. Each server in KSMS extracts part of the car identity and key share related to the owner by securely comparing the car ID stored in their database with the ones in the booking details provided by the car owner.

Access Token Generation Phase: The KSMS servers uses two consumer-generated symmetric keys for access token encryption. Using the previously extracted car key and booking details, the individual servers generate the encryption and MAC keys of these mutually agreed booking details using car key shares and the consumer's

symmetric keys. This is achieved by using a multiparty symmetric encryption protocol. The encrypted access token as well as the MAC of the token are then posted on the PL.

2.1.2.3 Security and Privacy Properties

The following security properties are satisfied by SePCAR [48, 17]:

- Confidentiality of booking details: only the owner and consumer have access to the sensitive information stored in booking details.
- Confidentiality of access token: only the car and the consumer are aware of the information stored in the access token.
- Confidentiality of car key: only the car and car manufacturer have access to the car key.
- Authenticity of booking details: the integrity, origin and approval of the booking details is checked by the car.
- Backward and forward secrecy of the access token: compromise of a key that is used to encrypt an access token should not lead to the compromise of any other access token published on the public ledger.
- Non-repudiation of the origin of access token: the owner should be able to prove the authenticity of the access token to car and consumer.
- Non-repudiation of delivery of access token: the consumer proves to the owner that the car has received the delivery of the access token.

Along with the security properties listed above, SePCAR also satisfies the following privacy properties [48, 17]:

- Unlinkability of consumer and shared car: multiple booking requests of the same consumer and car cannot be linked by anyone except the owner, consumer and the car.
- Anonymity of consumer and car: the real identity of the consumer and the car is not known by anyone except the owner, the consumer and the car.
- Undetectability of access token operations: the operations on access token such as its generation, update and revocation cannot be distinguished by anyone except the owner, consumer and the car.
- Forensic evidence provision: In an adverse scenario, the authorities should be able to retrieve forensic evidence from the system and gain access to the car. There should be no compromise in the privacy of any other user in such a scenario.

2.2 Blockchain & Cryptocurrency

2.2.1 Blockchain

The blockchain technology has a variety of definitions on the internet. As David Gerard, the author of *Attack of the 50 Foot Blockchain: Bitcoin, Blockchain, Ethereum & Smart Contracts* [27] said in an email to *The Verge*, "What is a 'blockchain'? The word is a buzzword that is increasingly ill-defined," it clearly means that the term blockchain has become so widely used, that it has lost its original meaning. A very well known cryptocurrency, Bitcoin is frequently used to define blockchain, however this definition fails to define many systems that are classified as blockchains. As for alternate definitions, *Forbes* defines blockchain as a public register that saves transactions commenced between two users in a secure, verifiable and permanent way. The data of these transactions is stored in cryptographic blocks which are connected to other blocks in a hierarchical manner [28]. *Wikipedia* defines blockchain as a decentralized, distributed and public digital ledger used to record transactions across multiple computers to prevent these records from being altered without altering the records in all other subsequent blocks in the network [56].
The problem with the above definitions is that all blockchains are not recorded publicly, as can be seen in the case of Walmart [30] where they are trying to implement a private blockchain, and many others are not decentralised. Some would even argue that a blockchain need not be digital. A wide definition that covers almost all implementations of blockchain can be as follows: A distributed computing architecture where each participating computer in the blockchain network is called a node. Each of these participating nodes have information about all historical transactions and the information these transactions shared. Grouping of these transactions is done into blocks which are added to the distributed database in a successive manner. At a time only one block is verified with the help of a mathematical proof and added. The verification is done to make sure that the newly added block follows in sequence from the previous block, ensuring that these blocks are connected in a chronological order [6]. The blockchain technology described above has established the era of 'digital' economy, it consists of seven design principles [51] detailed below:

1. Networked Integrity
   All the users of the network accept and verify each transaction and record it cryptographically on the blockchain. Each participant can exchange value with a mutual expectation of integrity from any other party and no one can hide these transactions. Each block must verify the preceding blocks, hence integrity is encoded into every step of the process and distributed amongst all members of the blockchain.

2. Distributed Power
   Blockchain consists of a peer-to-peer network and power is distributed amongst all these peer's equally. The blockchain can not be shut down by a single entity, and even if a central authority manages to shut down part of the blockchain,

the whole system would still be operational. All participants of the blockchain are aware of each transaction being carried out.

3. Value as incentive
   Blockchain technology has inbuilt incentives for participants who ensure its continuity and integrity. In the case of Bitcoin, each miner works in order to create new blocks and link them to previously created blocks, and the incentive for this collective task is a reward of some bitcoins to each participant. The incentives of all stakeholders are aligned in the same direction.

4. Security
   Cryptography is a must-have for participants of the blockchain. In the paper "Bitcoin: A Peer-to-Peer Electronic Cash System" [39], of Satoshi Nakamoto, participants are required to use a public key infrastructure (PKI).

5. Privacy
   Participants of the blockchain choose their own degree of anonymity. It is their own choice to add any personal details to their identity, hence they can remain anonymous. Blockchain also provides a separation of the transaction layer from the identification and verification layers hence making the blockchain somewhat private.

6. Rights Preserved
   The blockchain serves as a public registry using a tool called Proof of Existence (PoE). All ownership rights are transparent and enforceable and freedom of all participants is recognised.

7. Inclusion
   The system designed by Satoshi Nakamoto can even work without internet, where a typical person interacts with the blockchain through 'simplified payment verification', lowering the cost of transmitting funds. This is a solution for current modes of payments where a person uses a mobile application that requires phone cameras and QR codes, which incur extra costs to operate.

2.2.1.1 Types of blockchain

There are three types of blockchain from a technological perspective, namely Private, Public and Consortium blockchain. They are described as follows:

- Private Blockchain
  In a private blockchain, no one except a central organisation can do write operations on the blockchain. However, the read operations are allowed to be done by everyone or a restricted subset of the population. Applications of such blockchains can include Auditing and Database Management, as they are usually an internal operation of a company [11]. The example of Walmart can be used here, as they are trying to shift from a supply chain model to a private blockchain [30].

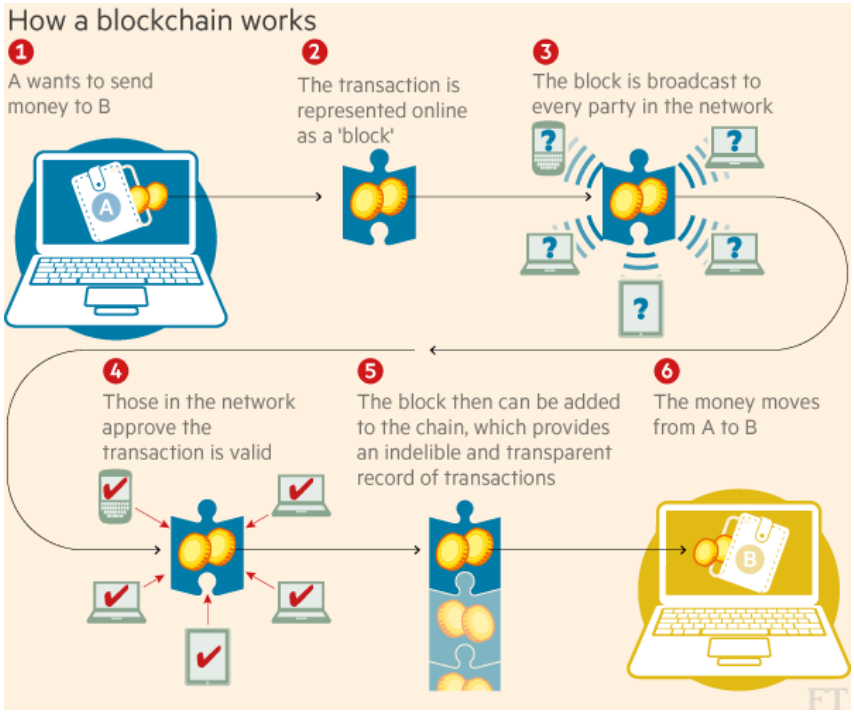

FIGURE 2.2: How Blockchain Works. [14]

	Public Blockchain	Private Blockchain	Federated/Consortium Blockchain
Access	• Anyone	• Single organization	• Multiple selected organizations
Participants	• Permissionless • Anonymous	• Permissioned • Known identities	• Permissioned • Known identities
Security	• Consensus mechanism • Proof of Work / Proof of Stake	• Pre-approved participants • Voting/multi-party consensus	• Pre-approved participants • Voting/multi-party consensus
Transaction Speed	• Slow	• Lighter and faster	• Lighter and faster

FIGURE 2.3: Classification of Blockchains and their Properties. [52]

- Consortium Blockchain
  A consortium blockchain is a partly-private blockchain in which a pre-selected set of nodes handle the consensus process, which is the operation where each node accepts and verifies a block before it becomes a part of the blockchain. The write operation is only limited to these pre-selected nodes, whereas the read operations may be public or restricted to a certain population [11]. Consortium blockchains are known to provide similar kinds of benefits as their private

counterparts while distributing the power to a group of members and not providing it to only one entity.

- Public Blockchain
  The blockchains that allow everyone to perform read and write operations on the blockchain are called Public Blockchains. Anyone can send transactions to these blockchains and these transactions are included if they are valid. Unlike the Consortium blockchain where the consensus process is controlled by a pre-selected set of nodes, everyone takes part in the consensus process of public blockchains. They are meant to replace centralised trust by using the concept of cryptoeconomics, which is the combination of monetary incentive and cryptographic verification [11]. Bitcoin is an example of public blockchain.

2.2.1.2 The Ethereum Blockchain

The Ethereum blockchain is a consortium blockchain that enables people to develop and deploy their own decentralised applications. It also offers a Turing-complete programming language, which enables anybody to develop smart contracts and decentralised applications [10].

1. Ethereum Accounts

   Each state in Ethereum has 'accounts', and each account has a 20-byte address. It consists of four fields [10]:

   - Account Nonce - the transaction count of an account .
   - Current ether balance, where 'Ether' is the intrinsic currency charged for transaction execution.
   - The contract code
   - Account's storage

2. Messages and Transactions

   Messages are produced by either a foreign entity or the contract itself and they optionally can contain data and can be responded to by the recipient, which shows that the concept of functions is embedded in Ethereum. The data packages that are used to store these messages are called 'transactions' which hold information about the message recipient, sender's signature, total ether and data to be sent. Along with these fields, transactions also consist of two values, namely, STARTGAS and GASPRICE. STARTGAS can be defined as the limit to the amount of computation steps of code execution a transaction can have, and each computational step incurs a fee to be paid to the miner, which is called GASPRICE [10] .

3. Ethereum State Transition Function

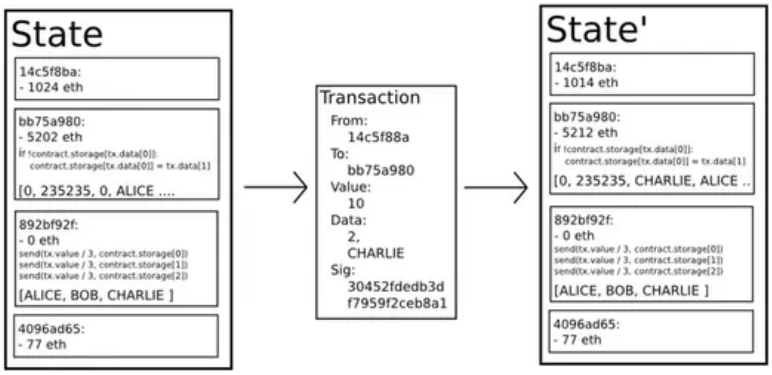

FIGURE 2.4: State Transition in Ethereum. [52]

The function used to transfer value and information between accounts is known as the Ethereum state transition function. It can be defined as follows:

- Preliminary checks are done, such as, checking if the transaction contains necessary values, validity of the signature and if account nonce is correct.
- The transaction fee (STARTGAS*GASPRICE) is deducted from the sender's ether balance and the account nonce value is incremented. An error is returned if the sender does not have the needed balance in his account to carry out the transaction.
- The GAS value is initialised to STARTGAS and a certain value of gas per byte is paid for the total number of bytes in the transaction.
- The transaction value is transferred from the sender to the receiver. All changes are reverted if the transaction fails. In case all goes well, the fees for remaining gas is returned to the sender's account and the fees for gas consumed is paid to the miner. The miners get their incentive irrespective of the outcome of the transaction.

4. Applications

As listed by Vitalik Buterin [10], there are three types of applications on top of Ethereum:

- Financial applications - hedging contracts, savings wallets, wills.
- Semi-financial applications - Applications where money is involved to carry out a non-monetary task, like bounties for solving complex math problems.
- Non financial applications - Decentralised authorisation, voting.

2.2.2 Cryptocurrency

Cryptocurrency is a term given to the digital form of money, it is a decentralised monetary digital asset which uses cryptography to ensure secure exchange or transactions. Since there is no central authority governing these cryptocurrencies, it uses a peer-to-peer network where every participant has equal authority and the validation of each transaction is done by each peer before it is accepted. The first widely accepted cryptocurrency is called Bitcoin, which was launched in 2009, and after that a lot of 'altcoins' surfaced that represent a mix of Bitcoin alternatives. As it is illustrated in Figure 2.5, these cryptocurrencies are rising in value everyday, sometimes facing a lot of fluctuation in their values, and this can be credited to a lot of technological and economical factors such as:

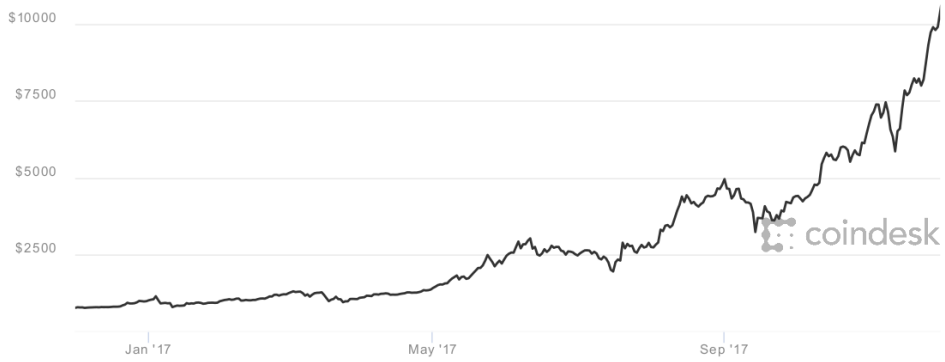

Figure 2.5: Change in Value of Bitcoin (in USD) in 2017. [13]

1. Technological factors
    - Decentralized – it is not controlled by any central authority or individual. It is corruption free and difficult to be manipulated.
    - Digital Currency – next generation of money and has tremendous utility.
    - Deflationary in nature.
2. Economical factors
    - Overall Global Financial Crisis [9].
    - Rising Inflation.
    - Brexit, Demonitisation in India [9].

FIGURE 2.6: Rising Hash Rate of Bitcoin. [8]

2.2.2.1 Bitcoin

The mechanism of cryptocurrencies can be explained using Bitcoin as an example, like any other cryptocurrency it uses a peer–to–peer network where each peer has a record of all historical transactions that are logged into previous blocks. When a transaction is made, it is signed using the private key of the person making the transaction and then broadcasted to the whole network. These transactions and their amounts are then logged in a new block which is verified and then connected to the blockchain. The verification and connecting of the new blocks to the blockchain is done through a process called 'mining'. Each peer connected to the blockchain performs this mining operation which requires high computational power [40], as mining requires solving a complex math problem. Figure 2.6 shows the increase in the hash rate of the Bitcoin blockchain. Hash rate can also be interpreted as the processing power being used to solve these complex math problems. The increase in this hash rate denotes the increase in the number of entities performing these mining operations. The reason behind such an exponential increase in the interest of people to perform mining operations is monetary incentive. The entities that perform these mining operations are called 'miners' and each miner gets an incentive of a few bitcoins when a new block is verified and connected to the blockchain, hence the more blocks mined, the more money earned. With the increase in the numbers of these miners, the difficulty of the math problems to be solved also increases, hence requiring an increase in the computational power.

2.2.2.2 Ethereum

In addition to Bitcoin, there are various other cryptocurrencies that are currently present in the digital market, such as, Ether. The Ethereum blockchain has its own cryptocurrency called 'Ether', also referred to as cryptocurrency 2.0 because it takes the underlying characteristics of Bitcoin and applies it to provide a wide variety of new applications. Although the value of Ether is very volatile, the main value of Ethereum is driven with its capabilities of completely eliminating third party intrusions in contractual obligations. In order to make use of these utilities provided by Ethereum, people have to use Ether. Figure 2.7 shows a comparison between Ethereum and Bitcoin.

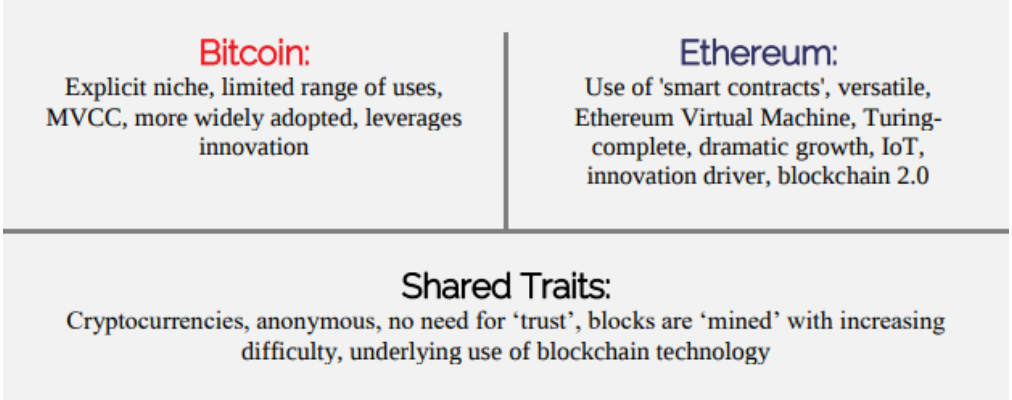

Figure 2.7: Comparison between Bitcoin and Ethereum. [20]

2.3 Smart Contracts

Smart contracts are neither smart, nor contractual by nature. They can be defined as contractual constraints deployed on the blockchain that act as an immutable agreement, and can receive or execute transactions. This term was coined by Nick Szabo, and it can be defined as a computerised protocol that executes the terms of a contract with the objectives of satisfying common contractual conditions, minimising malicious and accidental incidents, while not relying on trusted third parties. Smart contracts also have economic goals such as lowering transactional and enforcement costs, and minimising loss to fraudulent activities [50]. According to Szabo, the characteristics of a true smart contract are:

- Observability – The participants of the smart contract should be able to see how well their counterparts abide by the terms.
- Online Enforceability – Ensure that the agreed terms are being fulfilled by using various measures.

- Verifiability – In case a conflict arises between the participants, the contract should be auditable.
- Privity - Data of the smart contract should only be available to the participants of the smart contract, otherwise encrypted.

One important aspect is that the aforementioned characteristics of a true smart contract oppose each other. While characteristics like Observability, Verifiability and Online Enforceability enforce the transparency of the contractual data, Privity on the other hand requires the minimisation of such transparency. Thus, smart contracts should be developed with the optimum mixture of these characteristics, requiring it to be selectively transparent or private.

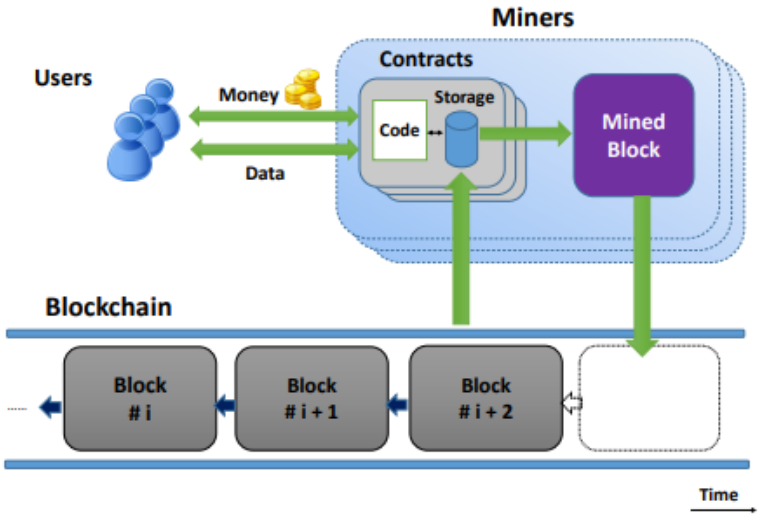

FIGURE 2.8: Execution of a Smart Contract on Blockchain. [16]

As shown in Figure 2.8, the public blockchain holds the storage file of the smart contract where a network of miners execute its logic and update the blockchain by reaching a consensus. The execution of the contract's code starts when it receives a message and while execution it writes or reads from its storage. Smart contracts are capable of receiving money from other contracts or users and sending money to them. The entire state of a contract is visible to the public [16].

For our implementation, we used the Ethereum Virtual Machine (EVM) to execute smart contracts.

2.3.1 Solidity

Smart contracts are developed using a Turing-complete scripting language called **Solidity**, after developing the smart contract it is compiled into EVM byte code and then deployed on the blockchain. Since smart contracts possess monetary value, there are common guidelines for developing a smart contract in Solidity, namely:

- Damage Control – There should be a limit on the monetary value stored in a smart contract.
- Modularity – Modular smart contracts are easy to improve and their length should be kept compact.
- Check-Effects - Preconditions should always be implemented before function execution that leads to state-changes.

1. **Variables & Data Types**
   In Solidity, variables are used to permanently store information in the contract's storage. These variables have several elementary data types, listed below [23]:
   - Booleans - used to store boolean values (T/F).
   - Integers - has two variations, namely, signed (int) and unsigned (uint) integers differing in byte lengths.
   - Address - it is used to represent the 20-byte address of an Ethereum account.
   - Byte arrays - has two variations, fixed-size which range in length from length 1 - 32, and dynamically-sized that are defined by 'bytes' type.
2. **Function Calls**
   Executable units that are capable of being called internally or externally are called Functions. Internal function calls are defined as the calls to a function in the current contract and external function calls are defined as calls to a function in an external contract. An example of a function call from our implementation is illustrated in Code Snippet 2.1. This function will be explained in Chapter 3.

```
1    function nonAccessWithdrawal(address _user) public carReady{
2        assert(_user == currentDriverAddress);
3        assert(canAccess==false);
4        clientBalance = ownerDeposit + clientDeposit;
5        msg.sender.transfer(clientBalance);
6        ownerBalance = 0;
7
8    }
9
```

Code Snippet 2.1: Function to penalise a fraud owner

3. **Visibility of Functions and State Variables**
   Solidity has four different types of visibility's for functions and variables, namely, public, private, external and internal. These are explained as follows [23]:
   - Public - functions that can be called internally and externally.
   - Private - functions and variables that only have internal visibility.

- External - these functions behave like public functions but can only be invoked by using 'this' keyword.
- Internal - these functions behave like private functions but cannot be called using 'this' keyword.

4. **Function Modifiers**
   Function modifiers modify the behaviour of functions and ensure that functions can only be executed when specific conditions are met. An example of function modifiers from our implementation is illustrated in Code Snippet 2.2.

```
1         modifier ifClient() {
2             if(client != msg.sender){
3             assert(false);
4             }
5             _;
6         }
7
```

Code Snippet 2.2: modifiers in solidity

5. **Events**
   Events allow the usage of logging facilities provided by the Ethereum Virtual Machine (EVM). Applications use Events to oversee a contract's state without interaction [23]. An example of Events from our implementation is illustrated in Code Snippet 2.3

```
1     event E_RentCar(address indexed _currentDriverAddress, uint
      _rentValue, uint _rentStart, uint rentEnd);
2
```

Code Snippet 2.3: Events in Solidity

2.4 Existing Work on Blockchain & Smart Contracts

Biswas and Muthukkumarasamy [7] propose a system that provides a secure communication platform between citizens and the government in a smart city using blockchain technology integrated with smart devices. Knirch et al. [31] proposes a protocol which uses blockchain technology and privacy-preserving smart contracts to make tariff decisions in smart grids by selecting the best-matching tariff for a customer with high accuracy. Al-Bassam [3] provides a solution to the problem of targeted attacks using rogue certificates by proposing a Public Key Infrastructure (PKI) system based on decentralised and transparent design using a smart contract on Ethereum blockchain. Zyskind, Nathan and Pentland [58] propose a decentralised privacy-preserving and secure personal data management system that recognises users as the owners of their own personal data and are always aware of what data is being collected. Aitzhan and Svetinovic [2] have designed a proof of concept

for secure and private decentralised smart grid energy trading based on blockchain technology which enables anonymous negotiation of prices and secure trading.
Systems that hold highly sensitive information are also adapting to the blockchain technology, for instance medical records of individuals, and Azaria et al. [4] applies the principles of decentralisation to large scale data management in an Electronic Medical Record (EMR) system that securely handles medical records and enables various operations, for instance, auditing them while keeping them private. When we consider application scenarios that adapt blockchain technology to car sharing systems, the two most successful real-time applications are La'Zooz [33] and Arcade City [15].
La'Zooz, publicly known as the 'blockchained version of Uber' is a decentralised ride-sharing network which enables private car-owners to share their empty seats with other people. It can also be called ride-sharing 2.0, as it elevates the whole system to a decentralised transportation network that eradicates issues like surge pricing, privacy leaks. The working mechanism of La'Zooz is explained as follows:

- The smart phones and computers of users are registered as 'road miners' that are main computing nodes in the blockchain.
- The real-time data is then stored in a cryptoledger using which operations like scheduling, payments are executed.
- La'Zooz uses a 'Proof–of–Movement' mechanism that gives an incentive of 'zooz' tokens to road miners who simply have to drive their car while La'Zooz's decentralised application (Dapp) runs on their smart phones or computer. This incentive scheme helps La'Zooz to build their local transportation web [33].
- La'Zooz is based on the critical-mass scheme, which means that it only activates it services in an area when their is a critical mass of users present. It has integrated algorithms that sense this presence of critical mass and activate the services in that area.

We know of only one effort involved in implementing a smart contract for booking and payments of private cars notably known as "A Smart Contract for a Smart Car" [45], though it has a lot of limitations in functionalities of the contract, and a lack of security and privacy of entities involved.

2.5 ECIES

If the participants of the smart contract send their sensitive information in plain-text, this information will eventually be broadcasted throughout the entire blockchain network. Since the participants might be selectively anonymous, they are reasonably expected to behave dishonestly and selfishly in order to gain more money. Hence, our implementation needed cryptographic primities such as encryption to defend our participants from fraud and to make sure their sensitive information is protected.

An encryption scheme that consists of a pair of keys, where the one used for encryption or verification of signatures is called Public key and the other that is used for decryption or performing signatures is called Private key, is known as Public-Key Cryptography. One of such Public-Key algorithms that is based on the algebraic structure of elliptic curves over finite fields is known as Elliptic Curve Cryptography (ECC). Using ECC, efficient and compact cryptographic keys can be made which are as secure as RSA (one of the first public-key cryptosystems), Table 2.1 shows the difference in bit lengths of RSA and ECC. One of the several protocols that have been adapted to Elliptic Curves is called Elliptic Curve Integrated Encryption Scheme (ECIES) which is the most popular encryption scheme based on Elliptic curves [37].

TABLE 2.1: Bit Length Comparison between ECC and RSA. [1]

ECC (bits)	**RSA** (bits)	**Key Size Ratio**
160	1024	1:6
256	2048	1:8
384	7680	1:20
512	15360	1:30

2.5.1 A Brief History of ECIES

Diffie-Hellman Integrated Encryption Scheme (DHIES) is the name given to the improved iterations [43] [44] of the original work done by Phillip Rogaway and Mihir Bellare known as "Discrete Logarithm Augmented Encryption Scheme (DLAES)"[42]. DHIES extends ElGamal encryption scheme and includes the following:

- Public key operations - cryptographic systems that consist of a pair of keys called Public and Private key.
- Symmetric encryption algorithms - algorithms which use the same key for both encryption and decryption.
- Message Authentication Code (MAC) functions - generate a tag which is used to authenticate messages.
- Hash computations - takes the message as an input and returns an unreadable alphanumeric string.

DHIES uses elliptic curves in an integrated scheme and requires less number of slow elliptic curve operations to be secure against chosen ciphertext attacks which makes it more lucrative. [44]. ECIES is nothing but DHIES restricted to Elliptic Curve group.

2.5.2 Functional components

ECC fits with our implementation as it is used mainly for security applications where integrated circuit space and computational power is limited, such as smart cards and wireless devices. ECIES has key components that it uses in the encryption and decryption process, described as follows [36]:

- Key Agreement (KA): This function is used to create a shared secret value.
- Key Derivation Function (KDF): Uses the secret computed by KA to generate the Encryption and MAC keys.
- Hash: Computes the message digest to preserve its integrity.
- Encryption (ENC): Used for encrypting the plain-text message and decrypting the encrypted message.
- MAC: Authentication of the message is done using the MAC function.

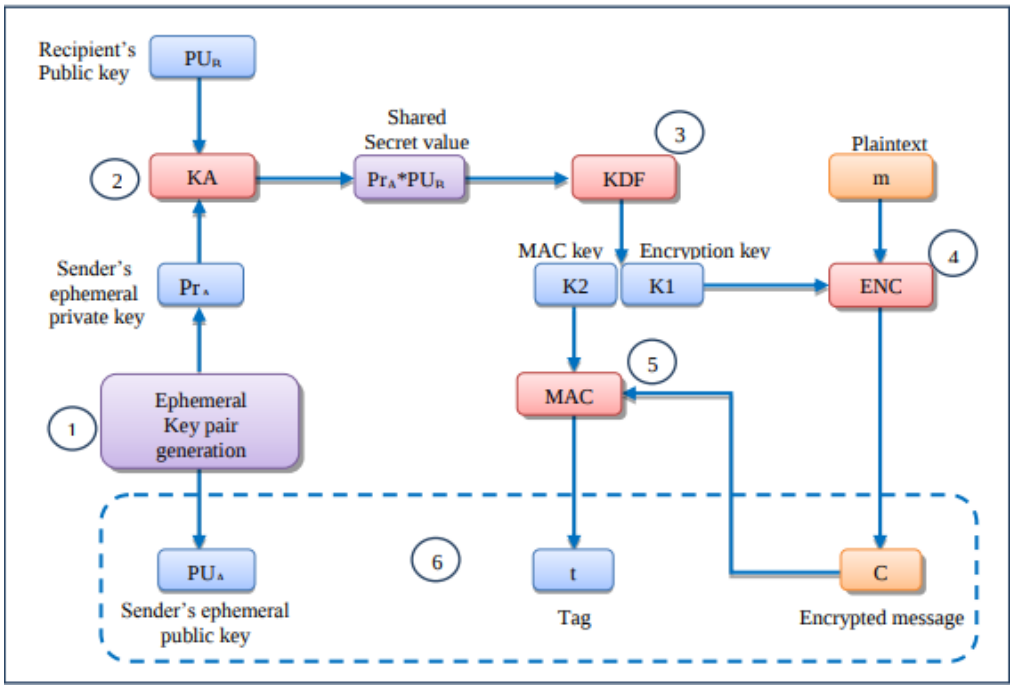

Figure 2.9: ECIES Encryption Process. [1]

Figures 2.9 and 2.10 depict the ECIES encryption and decryption process respectively. Let's consider two users, Ann and Ben and there is a scenario in which Ann wants to send a message to Ben. Ann has a pair of keys (private, public) represented as PRI_A and PUB_A, respectively, whereas Ben's keys are represented as PRI_B (private key) and PUB_B (public key). The steps (shown in Figure 2.9) that Ann must complete are the following [37]:

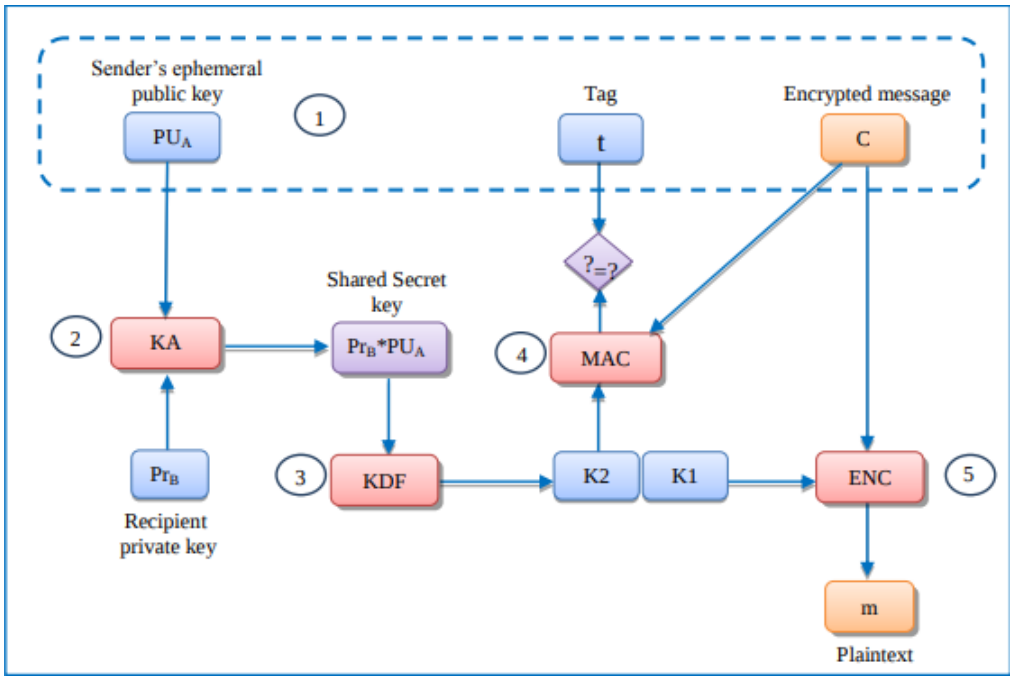

FIGURE 2.10: ECIES Decryption Process. [1]

1. Ann must create a pseudorandomly generated key pair $K_p = PRI_A \cdot PUB_A$.

2. Ann then uses the KA in order to create a shared secret value, $PRI_A \cdot PUB_B$.

3. This shared secret value is then pushed as input for KDF, which then outputs the symmetric encryption key (k_{ENC}), and the MAC key (k_{MAC}).

4. Using the Encryption Key (K_{ENC}), Ann uses the symmetric encryption algorithm, ENC, to produce the encrypted message (c) from the plaint-text message (m).

5. Then a tag is produced by selecting a MAC function and using k_{MAC} and the previously encrypted message as inputs.

6. Finally, the cryptogram (PUB_A||tag||c) is sent to Ben.

The following steps illustrate the decryption process that Ben must perform (see Figure 2.10) [37]:

1. Ben initially retrieves PUB_A, the tag and the encryped message (c) from the cryptogram.

2. Then he uses the previously retrieved PUB_A, and PRI_B to produce a shared secret value $PRI_B \cdot PUB_A$.

3. Taking as input the shared secret value $PRI_B \cdot PUB_A$, Ben uses KDF to produce similar encryption and MAC keys as Ann did.

4. Ben then generates a tag* using k_{MAC}, the encrypted message c, and if its different than the tag produced before, Ben must reject the cryptogram due to a failure in MAC verification procedure.

5. If the tag value generated by Ben is the correct one, he is able to decipher the encrypted message by using the symmetric ENC algorithm and k_{ENC}.

Chapter 3

Smart Contract for Secure Car Sharing

In this chapter we discuss our implementation in detail along with all case studies applied while developing it.

3.1 Overview of the Smart Contract

In the following points, we will give a high level description of how our smart contract works:

- The owner of the car deploys the smart contract on the blockchain and deposits a value of 5 ether in it.
- The customer (person who wants to rent the car) sets the number of days he wants to rent the car and sends a request to rent the car by depositing 5 ether in the deployed smart contract.
- The owner and the consumer come to a mutual agreement on the booking details like the pick-up location and price per day.
- Once the details are discussed, the owner signs the booking details using his private key and encrypts them using the customer's public key. Once encrypted he sends these details to the storage of smart contract and allows the customer to access the car. The encrypted details stored in the smart contract can be accessed by the customer and decrypted using their private key.
- The customer accesses the car and uses it for the decided amount of time. When their trip is over, the customer ends the car rental process by dropping off the owner's car at the discussed location.
- Once the process is over, both the owner and the customer can separately withdraw their deposits and earnings.

The description given above is an overview of an ideal scenario of the rental process. There might be some situations where entities are malicious, but our smart contract is designed to counter this behaviour.

3.2 Design Requirements

This section will highlight the design requirements of our smart contract. In our implementation, we tried to link our smart contract to the booking and payments aspect of SePCAR [48], hence we list the security and privacy requirements of SePCAR and how our smart contract fulfills them, along with the functional requirements that arise in a secure booking and payments system.

3.2.1 Security and Privacy Requirements

An overview of the security and privacy requirements of SePCAR is given in Chapter 2. Our implementation satisfies the following security requirements of SePCAR [48]:

- Confidentiality of booking details: only the owner and the customer have access to the sensitive information stored in booking details, as these details are discussed offline and only stored in the smart contract when encrypted.
- Confidentiality of access token: the encrypted booking details are treated as the access token and these details can only be decrypted by the customer.
- Authenticity of booking details: owner's signature on the booking details is verified before giving access to the customer.
- Non-repudiation of the origin of access token: the owner signs the access token with his private key to prove the authenticity of his identity and the access token.

Along with the security requirements, our implementation also satisfies the following privacy requirements of SePCAR [48]:

- Anonymity of consumer and car: on the blockchain, the consumer can choose to be anonymous and only interact using his 20 byte address, and the identity of the car is encrypted by the owner before storing it on the smart contract.
- Forensic evidence provision: smart contracts are auditable by nature, and in case of an incident involving the owner or the consumer, our smart contract can be audited to reveal information about them.

3.2.2 Functional Requirements

In the first section we discussed an ideal scenario where both, the customer and the owner, were honest entities. However this is not always the case and our smart contract needed to tackle adverse situations, hence we aim to achieve the following functional requirements from our final smart contract:

- Carry out the whole booking and payments process in a secure way.
- In case the customer is a fraud, penalise them and return the total deposit to the owner and vice versa in case the owner is malicious.
- Implement a proper cancellation functionality. Only give access to the customer once the owner allows it.
- In case of extra time taken by the customer, have a proper functionality to deduct the required extra amount from the customer's deposit and transfer it to the owner.
- Store the encrypted booking details so that the customer can access and decrypt them whenever required. Verify the authenticity of these details, the customer and the owner.

3.3 Methodology

As it is illustrated in Figure 3.1, we started from a basic case and continued on to the final smart contract using a modular approach. The flowchart shown below highlights the implementation phase of our thesis.

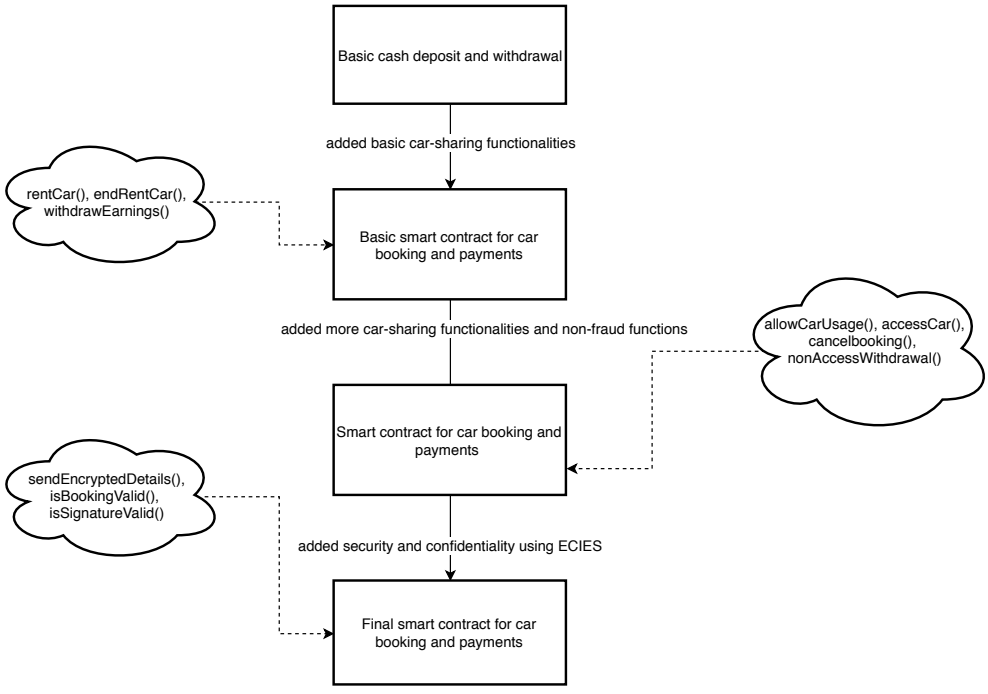

FIGURE 3.1: Final Smart Contract Development in Stages.

A list of the case studies developed is given below, along with a description and detailed explanation of each of the given cases. The developed case studies are as follows:

- Basic cash deposit and withdrawal
- Basic smart contract for car booking and payments - this case study meets the requirements of carrying out the whole booking and payments process in a secure way but still lacks other requirements.
- Smart contract for car booking and payments with extensions - along with carrying out the whole booking and payments process in a secure way, this case study also has extra functionalities like cancellation of booking, method to handle scenario's of extra time and also penalises the adverse entities.
- Final smart contract for car booking and payments with integrity and confidentiality - this case study fulfills all security, privacy and functional requirements.

3.3.1 Experimental Setting

All the case studies were implemented on the Ethereum blockchain. We use Ethereum, or also otherwise known as Blockchain 2.0, because of its ability to host decentralised applications. Ethereum comes along with a Turing-complete programming language known as Solidity that is used to write the smart contracts and as discussed in Chapter 2 allowing anyone to create their own arbitrary rules for ownership, in this case, the ownership of the car.

3.4 Design and Implementation

This section describes the design and implementation of each case study in detail.

3.4.1 Case study 1 - Basic cash deposit and withdrawal

In this case study we developed a basic smart contract that is able to carry out financial transactions such as cash deposit and withdrawal. The owner of the smart contract is capable of making such transactions by just calling a function for deposit and another for withdrawal. The idea behind developing this contract was to gain knowledge on making 'payable' functions that are used to make payments in the final smart contract. In this contract only the owner of the smart contract, i.e, the address from which this contract is deployed is allowed to withdraw money. On the other hand, everyone is allowed to deposit money. The functional flow of this smart contract can be seen in Figure 3.2.

3.4.1.1 Aim

This case study was developed with main idea of gaining knowledge of implementing payable functions in Solidity. Since our final smart contract is mainly concerned with

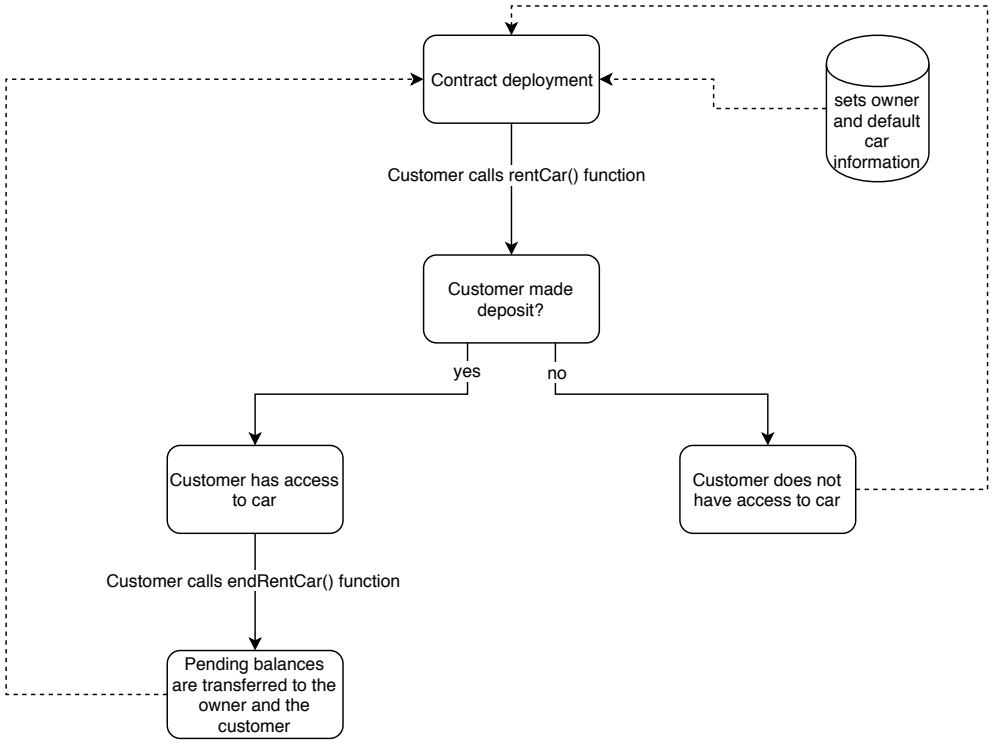

FIGURE 3.2: Initial Smart Contract Functional Flow.

the transactions involved in booking and payments, the ability to develop payable functions was needed. Along with gaining the knowledge about payable functions, we also learnt different ways of access restriction using a modifier that is used to modify function behaviour in Solidity.

3.4.1.2 Code explanation

```
1     modifier ifClient() {
2         if(client != msg.sender){
3             assert(false);
4         }
5         _;
6     }
```

CODE SNIPPET 3.1: modifiers in solidity

As mentioned in the above subsection, this modifier is used for access restriction in a smart contract. The function ifClient() is used to make sure that only the owner of the smart contract is able to withdraw money/get balance information. Having this modifier is a necessity as it provides trivial security to the owner of the smart contract, in the sense that only he/she is able to access their account balance or withdraw money as it verifies the address of the transaction caller with the address of the owner before calling these functions.

```
    function depositFunds() payable public{
        UpdateStatus('User transferred some money', msg.value);
    }
```

Code Snippet 3.2: Payable functions

The 'payable' keyword used in Code Snippet 3.2 enables the function depositFunds() to access the value property of the message that consists of the money being sent to different parties. This payable function is the core of the smart contract, as it enables everyone to deposit money to owner's account. The way this smart contract works is that, anyone can deposit money but only the owner has the ability to withdraw money.

```
    function withdrawFunds(uint amount) ifClient public{
        UpdateStatus('User transferred some money', 0);
        if(client.send(amount) || client.balance > amount){
            _switch = true;
        }
        else if(client.balance < amount){
            assert(false);
            _switch = false;
        }
    }
```

Code Snippet 3.3: Usage of modifiers

The modifier 'ifClient' in the above function restricts the access of this function to only the owner of the smart contract. Hence only the owner can withdraw the funds that were deposited by the various consumers of this contract, making it a safe cash deposit smart contract on the blockchain.

Although this smart contract implements cash deposit and withdrawal, we require a smart contract that can handle the booking and payments aspect of car rentals. In the next case study, we extend the functionalities of the current smart contract to handle the booking and payments aspect of a car sharing system.

3.4.2 Case study 2 - Basic smart contract for car booking and payments

In this case study we develop the smart contract that enables the owner of the smart contract to let any individual use his/her car on a rental basis for a fixed period of time. The customer has to deposit a certain amount of ether in the smart contract that is not accessible to the owner. When the consumer returns the car, the owner of the smart contract has to distribute the earnings by calling the distribute earnings functions. This function automatically computes the rent that is owed to the owner by the consumer, deducts it from the deposit and transfers it to the owner of the car, while the rest of the deposit is transferred to the consumer.

3.4.2.1 Aim

The aim of this case study is to first include all the basic functionalities of our final contract and test them on a real world snapshot of a private blockchain. In the

following case studies, this contract will be modified to include additional functional requirements.

3.4.2.2 Code explanation

```
1     constructor(SmartCar){
2         owner = msg.sender;
3         currentCarStatus = CarStatus.Idle;
4         currentDriverInfo = DriverInformation.None;
5         carIsReady = true;
6     }
```

Code Snippet 3.4: Constructors in solidity

A constructor is a function that is executed when the smart contract is deployed. As it is illustrated in Code Snippet 3.4 our smart contract has a constructor called, (constructor(SmartCar)), this constructor is used to set the crucial information regarding the smart contract, such as, the owner of the smart contract and the initial status of the car.

The above constructor assigns the address of the owner of the car to the smart contract which is stored in an address variable namely "owner". The initial status of the car is set to idle and since the car is available for rent, driver information is set to none. The different statuses of the car and its occupant are stored in Enumerations, which are a way to create user-defined types. Our smart contract has two enumerations, namely DriverInformation that stores the information about the current driver of the car and CarStatus that stores the information about the current status of the car.

After defining the various events, variables and the owner of the car, basic functionalities had to be implemented. We started with the basic functionality of renting the car.

```
1     function rentCar() public onlyIfReady payable{
2         if(currentCarStatus==CarStatus.Idle && msg.value==
      RATE_DAILYRENTAL){
3         currentDriverAddress = msg.sender;
4         currentCarStatus = CarStatus.Busy;
5         currentDriverInfo = DriverInformation.Customer;
6         currentDriveStartTime = now;
7         currentDriveRequiredEndTime = now + 1 days;
8         balanceToDistribute += msg.value - 500;
9         E_RentCarDaily(currentDriverAddress,msg.value,
      currentDriveStartTime,currentDriveRequiredEndTime);
10        }
11    }
```

Code Snippet 3.5: The rentCar() function

As it is illustrated in the Code Snippet 3.5, rentCar() function is a payable function that is used to start the car renting process. Preliminary checks are done before renting out the car, such as checking the current car status; if the car is ready to be rented. This function sets the current driver information to 'customer' so as

to make the owner unable to rent out the specific car to someone else. Along with that, the current driver address is changed to the address of the caller of the function (the consumer) and the current status of the car is changed to 'busy'. In addition to that, this function also calls the event 'E-RentCar' that starts computing the rental time. In order to call this function, the customer has to make a deposit to the smart contract, and the customer is allowed to rent the car only if the deposit made by him/her is equal to the money set by the owner. It is important to note that this deposit made by the customer can not be accessed by the owner of the smart contract. By calling this function, we initiate the car rental procedure.

```
1   function endRentCar() public onlyIfReady{
2       assert(currentCarStatus==CarStatus.Busy);
3       assert(currentDriverInfo==DriverInformation.Customer);
4
5       bool endWithinPeriod = now <= currentDriveRequiredEndTime;
6
7       emit E_EndRentCar(currentDriverAddress, now, endWithinPeriod);
8
9       currentDriverAddress = address(0);
10      currentCarStatus = CarStatus.Idle;
11      currentDriverInfo = DriverInformation.None;
12      currentDriveStartTime = 0;
13      currentDriveRequiredEndTime = 0;
14
15      distributeEarnings();
16  }
```

Code Snippet 3.6: The endRentCar() function

The endRentCar() function is used to end the car rental process. To call this function successfully, the car should be in 'Driving' state and the current driver information should be 'Customer'. This function also calls the E-EndRentCar event that checks if the car was returned within the allotted time. After the rental process is finished, the current driver address is set to null and the car information is set to default to enable the car to be rented again. We also call the distributeEarnings() function that calculates the amount that has to be deducted from the customer's deposit and given to the owner.

Along with the rentCar() and endRentCar() functions, we also have some functions that deal with the distribution of earnings, like the distributeEarnings() function and the withdrawal of the money. Withdrawal of money is handled by the withdrawEarnings() function that transfers the money from the smart contract to the accounts of both parties (owner and customer).

This implementation of the smart contract has the following limitations:

1. We assume the owner to be trustworthy. We make the owner the ultimate shot caller in this contract and the customer is deemed untrustworthy. There is no reward for the customer in case the owner is a fraud.

2. There is no functionality to cancel the booking, once made by the customer or agreed by the owner.

3. There is no computation of extra cost in case the customer takes extra time to return the car.

4. The customer can access the car as soon as he makes the deposit payment without any authentication by the owner.

All these drawbacks will be solved in our next case study that implements some additional functionalities to ensure the safety of both the customer and the owner.

3.4.3 Case study 3 - Smart contract for car booking and payments with extensions

To deploy the smart contract of this case study, the owner also has to make a deposit equal to the deposit he requests from the customer. This is done to ensure that in case the owner is malicious and does not respect the booking agreement, the customer gets the proper compensation. This deposit is not accessible by the customer until some fraudulent activity is done by the owner. We also implemented an 'extra time' functionality, and a fixed amount per extra day is deducted from the customer's deposit in case of extra time. Moreover, other functionalities will be explained with the help of snippets in further subsections. The functional flow of this smart contract can be seen in Figure 3.3.

3.4.3.1 Aim

The goal of this case study is to implement extra functionalities, such as penalising fraudulent activities and ensuring seamless, safe transactions between the parties involved in the contract. With this case study, both the owner and customer have an equal control on the smart contract which safeguards them from each other. The smart contract developed in this case study is a fully-functional smart contract aiming to handle all requirements of a car booking and payments system. Unlike the previous case study, money can only be withdrawn by both parties if there is a mutual agreement. In case of malicious users, the victim gets the whole amount stored in the smart contract. In this case study, we address a lot of limitations of the previous smart contract and provide added functionalities, such as:

1. Functionality for cancellation of booking.

2. Counter measures in case of fraudulent activities.

3. Computation of extra cost in case the customer takes extra time.

4. Owner has to allow the customer to access the car.

3.4.3.2 Code Explanation

```
    modifier clientAgrees{
        assert(clientReady);
        _;
    }

    modifier ownerAgrees{
        assert(ownerReady);
        _;
    }
```

Code Snippet 3.7: Extra modifiers to establish more control

In this case study we added extra modifiers to establish more control of both parties on the smart contract. The modifiers 'clientAgrees' and 'ownerAgrees' act as signatures by both parties, and money can be withdrawn from smart contract only if both the owner and the customer sign (mutually agree).

```
    function SmartCarMod() payable public{
        assert(msg.value == 5 ether);
        carOwner = msg.sender;
        ownerDeposit = msg.value;
        currentDriverInfo = DriverInformation.None;
        currentCarStatus = CarStatus.Idle;
        carIsReady = true;
    }
```

Code Snippet 3.8: Modified constructor smartCarMod()

As mentioned in the introduction of this case study, we modified our constructor making it mandatory for the owner to deposit the similar amount of money he asks from the customer. The 'assert' function makes it a mandatory task in order to deploy the contract. The rest of the functionality of the constructor is similar to the previous case study. We also added the payable keyword to enable the constructor to retrieve the value sent with the message using 'msg.value'.

```
    bool allowCarUse = false;
    function allowCarUsage(address _user) public carReady{
        require(_user == carOwner);
        allowCarUse = true;
    }

    bool canAccess = false;

    function accessCar(address _user) public carReady{
            require(_user == currentDriverAddress);
            require(allowCarUse);
            canAccess = true;
    }
```

Code Snippet 3.9: Functions to establish authenticated car access

The allowCarUsage() function requires the owner of the car to give explicit agreement that the customer can use the car. Unless the owner calls this function with his address, the customer can not access the car. This adds a layer of authentication so that the customer can not access the car without permission. We believe that it

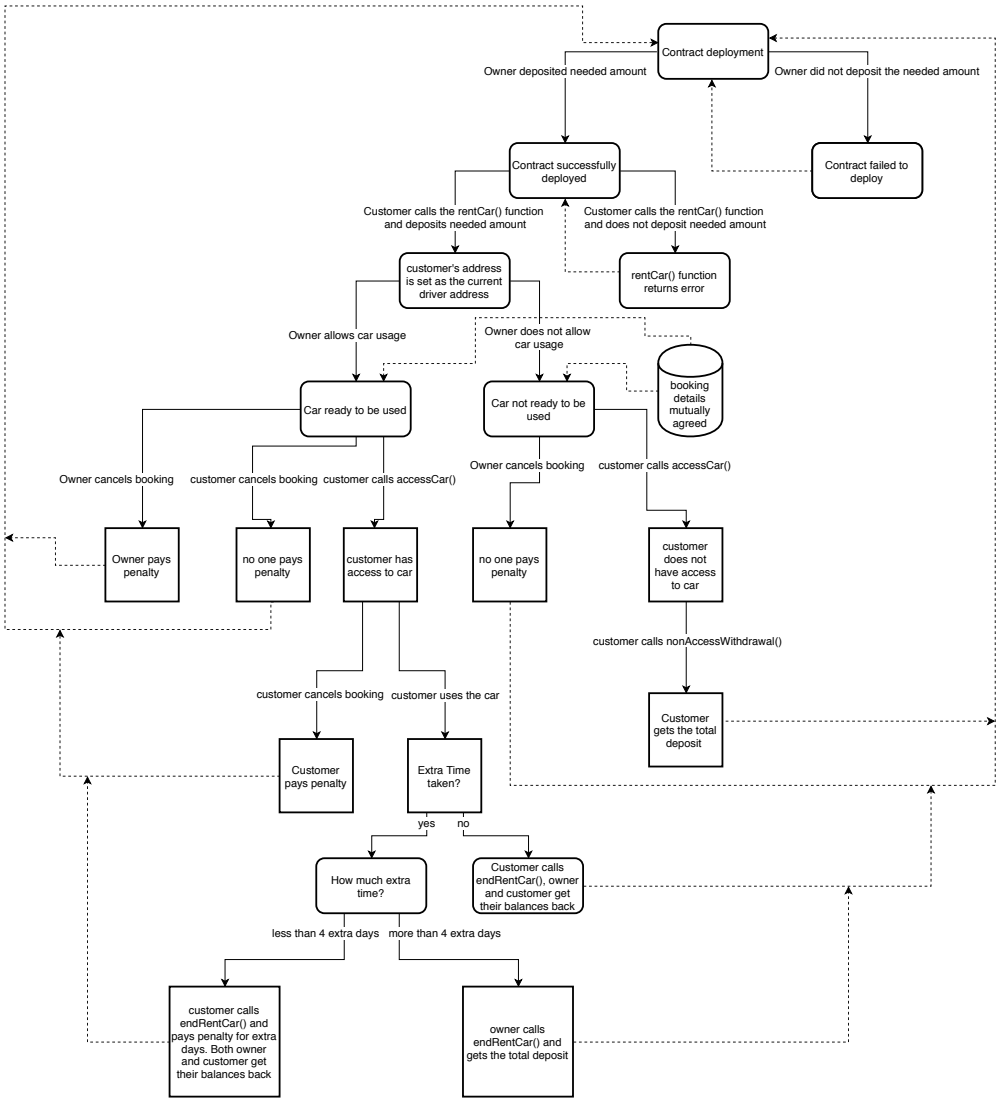

FIGURE 3.3: Functional Flow of the Smart Contract.

is important for the owner and customer to mutually agree on all booking details before the rental process starts and the customer can access the car. The accessCar() functions enables the customer to access the car, if and only if, the owner has already allowed the customer to use the car.

```
function nonAccessWithdrawal(address _user) public carReady{
        assert(_user == currentDriverAddress);
        assert(canAccess==false);
        clientBalance = ownerDeposit + clientDeposit;
        msg.sender.transfer(clientBalance);
        ownerBalance = 0;

}
```

Code Snippet 3.10: Function to penalise a fraud owner

With the improvements in this case study, as previously explained, we added functionalities to give access to the customer only if the owner explicitly allows it. However, the owner of the car can also be malicious. To avoid that, we implemented the nonAccessWithdrawal() function, shown in Code Snippet 3.10. Whenever the customer tries to access the car after having a mutual agreement with the owner and fails to do so, the smart contract penalises the owner of the car by transferring the total deposited money to the customer's account. This total deposit includes both, the owner's and the client's deposit.

```
function endRentCar() public carReady{
    assert(currentCarStatus == CarStatus.Busy);
    assert(currentDriverInfo == DriverInformation.Customer);

    balanceToDistribute = RATE - 3.5 ether;

    if(extraTimeTaken==true && (driveRequiredEndTime + extraTime) < 4){
        balanceToDistribute += extraTime * 500000000000000000; // 0.5 ether for each extra day
    }

    if(extraTimeTaken==true && (driveRequiredEndTime + extraTime) >= 4){
        assert(msg.sender == carOwner);
        E_EndRentCar(currentDriverAddress, now, false);
        clientBalance = 0 ether;
        ownerBalance = clientDeposit + ownerDeposit;
        msg.sender.transfer(ownerBalance);
        currentDriverAddress = address(0);
        currentCarStatus = CarStatus.Idle;
        currentDriverInfo = DriverInformation.None;
        driveStartTime = 0;
        driveRequiredEndTime = 0;
    }
    else
    {
        assert(msg.sender == currentDriverAddress);
        E_EndRentCar(currentDriverAddress, now, true);
```

```
27
28              currentCarStatus = CarStatus.Idle;
29              currentDriverInfo = DriverInformation.None;
30              driveStartTime = 0;
31              driveRequiredEndTime = 0;
32              clientReady = true;
33              ownerReady = true;
34              carFree = true;
35              distributeEarnings();
36          }
37
38      }
```

Code Snippet 3.11: Modified endRentCar() function

In this case study we also modified the endRentCar() function to add more functionalities like computing the extra time cost. The most important addition to this function is to prevent the owner from being a victim of scam, like we did by adding nonAccessWithdrawal() function for the customer. The basic understanding of this smart contract is that none of the involved entities are trustworthy and we implemented counter measures to prevent malicious behaviour. In the endRentCar() function, we added a condition that if the customer exceeds his/her borrowing limit by more than 4 days, we assume that the customer has fled with the owner's car and penalise the customer by transferring the total deposit to the owner. The compensation and deposits obviously have to be agreed by both parties to ensure a fair settlement. Code snippet 3.11 demonstrates the changes.

```
1       function cancelBooking(address _user) public carReady{
2           if(_user == carOwner && allowCarUse == false){
3               currentCarStatus = CarStatus.Idle;
4               currentDriverInfo = DriverInformation.None;
5               currentDriverAddress.transfer(clientDeposit);
6           }
7           else if(_user == currentDriverAddress && canAccess == false){
8               currentCarStatus = CarStatus.Idle;
9               currentDriverInfo = DriverInformation.None;
10              msg.sender.transfer(clientDeposit);
11          }
12          else if(_user == currentDriverAddress && canAccess == true){
13              currentCarStatus = CarStatus.Idle;
14              currentDriverInfo = DriverInformation.None;
15              msg.sender.transfer(clientDeposit - 500000000000000000);
16              carOwner.transfer(500000000000000000);
17          }
18          else if(_user==carOwner && allowCarUse == true){
19              currentCarStatus = CarStatus.Idle;
20              currentDriverInfo = DriverInformation.None;
21              ownerDeposit = ownerDeposit - 500000000000000000;
22              currentDriverAddress.transfer(clientDeposit +
    500000000000000000);
23          }
24      }
```

Code Snippet 3.12: Function that deals with cancellation of booking

The main limitation of the previous case study was that there was no way for either the owner of the car or the customer to cancel the booking. In this case study we implemented a cancelBooking() function, shown in Code Snippet 3.12, that deals with the cancellation situation. If the owner wants to cancel the booking and hasn't allowed the customer to access the car, he/she is allowed to cancel the booking without any penalty. However, if the cancellation is done after giving access to the customer, the owner has to pay a penalty to the customer. The case for the customer is identical, if the customer cancels the booking before accessing the car, there is no penalty on cancellation but if he/she tries to cancel the booking after accessing the car, some penalty is due.
There are still some limitations in this case study. For instance, what if the person posing as the owner is not actually the owner? Are the booking details actually agreed by the owner? If so, where is the proof? How can we encrypt the sensitive details so that they are not readable by everyone? How can we link each transaction with a person? All these limitations are solved in the next case study.

3.4.4 Case study 4 - Final smart contract for car booking and payments with security and confidentiality

This case study mainly focuses on the security and confidentiality aspect of the car booking and payments system. We try to bridge the gap between our smart contract and SePCAR [48] by fulfilling some of its security and privacy requirements, stated in section 2. We use JavaScript for the cryptographic side of our implementation and do the encryption and decryption off-chain for the sake of simplicity during verification of posting a block. This smart contract inherits the functionalities as described before, with an addition of sending parts of the booking details encrypted to the blockchain. As shown in Figure 3.4, booking details such as required number of days, price per day, price per extra day are not encrypted as they are used to compute the price of rental process. On the other hand, details like location of car, car plate number are sensitive data and can be encrypted. The functional flow of this smart contract can be seen in Figure 3.5

3.4.4.1 Aim

This case study aims to make parts of the booking details information confidential. The main essence of this case study is the use of public-key cryptography.

3.4.4.2 Code Explanation

The development in this case study was done in two phases. A program was developed in JavaScript for off-chain encryption and decryption, while some functions were added to the smart contract to support these cryptographic functionalities.

1. JavaScript program:
   Our JavaScript implementation handles the off-chain encryption, decryption and signature of transactions. We make use of several libraries to carry out

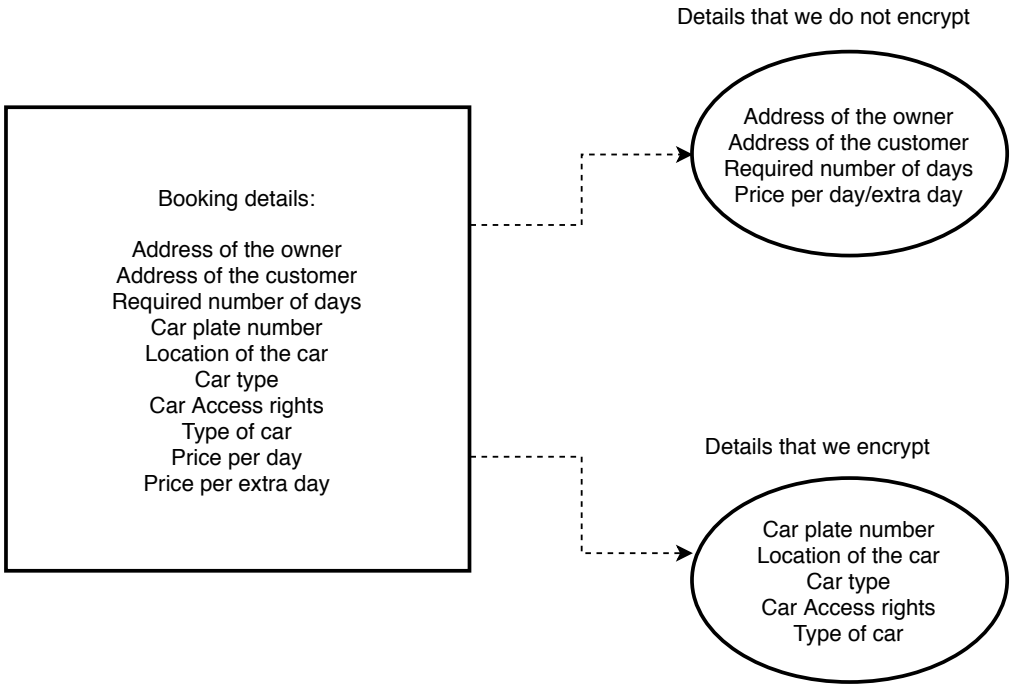

FIGURE 3.4: Encrypted and Plain-text Booking Details.

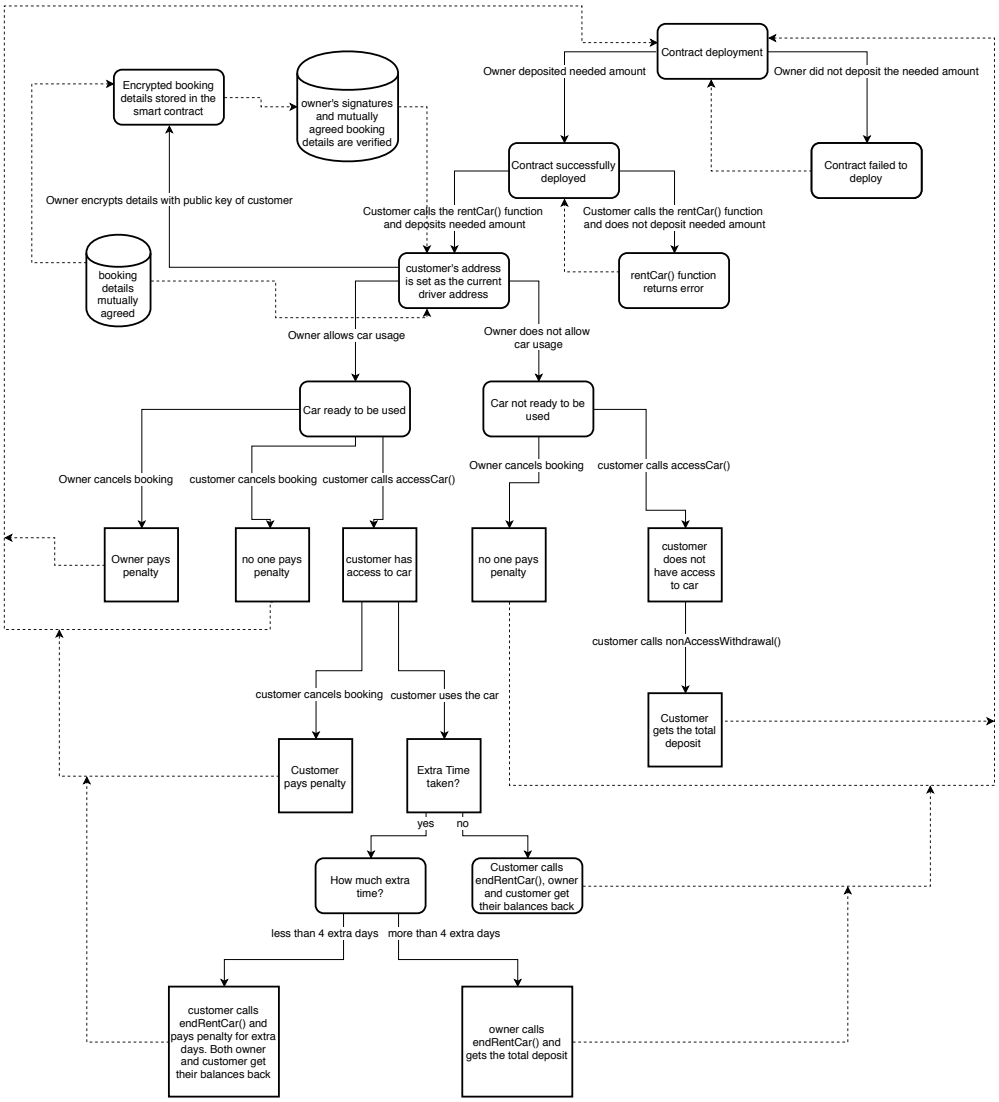

Figure 3.5: Functional Flow of the Final Smart Contract.

these transactions. Given below is a list of libraries that our JavaScript code makes use of:

- Eth-crypto - it is a library that consists of "Cryptographic JavaScript-functions for ethereum" [41] that can easily be used with web3 and solidity. We used this library to create the identities of our owner and customer which is comprised of their address, private key and public key. We also used this library to sign transaction with the private key of the entity executing the transaction.
- Ethereumjs-util - it is "a collection of utility functions for Ethereum" [25]. We used this library to retrieve the public key from a given private key in our JavaScript code.
- Eth-ecies - it is a library "for easy encryption using Ethereum keys" [35]. We used this library to perform the encryption and decryption operations off-chain.
- web3.js - This library "is a collection of libraries which allow you to interact with a local or remote ethereum node, using a HTTP or IPC connection." [24]. We use web3.js to deploy our contract on our local blockchain and to send the signed transactions to it.
- Ganache-core - This library provides a personal local blockchain for Ethereum development and testing. We use this to create a provider for our web3.js library in order to successfully connect to Ethereum. Ganache is a "Fast Ethereum RPC client for testing and development" [55].
- Solc - This library provides us "JavaScript bindings for the solidity compiler" [22]. We use this library to successfully compile our Smart contract because in order to deploy the contract web3 uses a compiled bytecode which is provided to us by solc.

```
1 const ownerIdentity = EthCrypto.createIdentity();
2 const customerIdentity = EthCrypto.createIdentity();
```

Code Snippet 3.13: Methods to create identities for owner and customer

As shown in Code Snippet 3.13, we make two separate identities for the owner and the customer. We need these identities as they provide us with the Ethereum keys of both entities and also their address using which they execute transactions.

```
1 const ganacheProvider = ganache.provider({
2     accounts: [
3         // we preset the balance of our ownerIdentity to 100 ether
4         {
5             secretKey: ownerIdentity.privateKey,
6             balance: web3.utils.toWei('100', 'ether')
7         },
8         // presetting the similar amount for our customer
9         {
```

```
            secretKey: customerIdentity.privateKey,
            balance: web3.utils.toWei('100', 'ether')
        }
    ]
});

// set ganache to web3 as provider
web3.setProvider(ganacheProvider);
```

Code Snippet 3.14: Initialising Ganache (Our local blockchain for Ethereum development)

In order to carry out transactions, both of our entities need to have some money in their accounts as each transaction costs some gas price, where gas is the internal pricing incurred when a transaction is executed. As illustrated in Code Snippet 3.14, we make a 'ganache provider' that initialises two accounts, each having a value of 100 ether, in the blockchain. We use this with web3.js to connect and carry out transactions with the Ethereum blockchain.

```
    const fs = require('fs');
    const solc = require('solc');
    let source = fs.readFileSync('SmartContractCrypto.sol', 'utf8');
    let compiled = solc.compile(source,1);

    const createCode = EthCrypto.txDataByCompiled(compiled.contracts[':SmartContractCrypto'].interface,
                                                  compiled.contracts[':SmartContractCrypto'].bytecode, [ownerIdentity.address]);

    //The deploy transaction, value is set to 5 ether so as to deploy the contract successfully.
    const deployTx = {
        from:ownerIdentity.address,
        nonce:0,
        gasLimit:5000000,
        gasPrice:500000000,
        value: parseInt(web3.utils.toWei('5','ether')),
        data:createCode
    };
```

Code Snippet 3.15: Compiling and deploying our smart contract on the local testchain using JavaScript

Code listing 3.15 shows the procedure to compile our smart contract using the 'solc' library in JavaScript. We need to compile our smart contract so as to successfully deploy it on the Ethereum blockchain because Ethereum requires a bytecode of the smart contract to initiate a transaction on it. The second part of the code snippet shows the 'deploy transaction' which requires a certain value in order to be successful, which in our case is 5 ether. Using the 'data' attribute we send the needed parameters for our transaction. Once our smart contract is deployed on the local testchain, we can execute all of its

functionalities, for instance, calling the rentCar() function. For example, we have given below a code snippet that calls the rentCar() function of our smart contract.

```
1    //Transaction to call rentCar() function. Requires a value of
     5 ether to be executed
2    const rentCar = contractInstance.methods.rentCar(
     customerIdentity.address, vrs.v, vrs.r, vrs.s, vrsDetails.v,
     vrsDetails.r, vrsDetails.s).encodeABI();
3
4    const rentTx = {
5        from:customerIdentity.address,
6        to:contractAddress,
7        nonce:1,
8        gasLimit:5000000,
9        gasPrice:500000000,
10       data:rentCar,
11       value:parseInt(web3.utils.toWei('5','ether'))
12   }
13   const serializedRecievedTx = EthCrypto.signTransaction(rentTx,
      customerIdentity.privateKey);
14
15   const receiptRentCar = await web3.eth.sendSignedTransaction(
     serializedRecievedTx);
```

Code Snippet 3.16: calling the rentCar() function using JavaScript

In Code Snippet 3.16 we initially create the Application Binary Interface (ABI) of our transaction and store it in the 'rentCar' constant. ABI is used to interact with Ethereum contracts either from outside the blockchain or through another contract in the blockchain [47]. We use the ABI to interact with the rentCar() function of our smart contract and send it necessary parameters by using the 'data' field. The rentCar() function also needs a value of 5 ether, hence we send it 5 ether from the account of the customer.

Using Code snippet 3.14, 3.15 and 3.16 we explain the interaction, compilation and deployment of our smart contract on the local blockchain using JavaScript libraries like solc, web3.js and ganache-core. Now we will focus on the main essence of this case study, i.e., the functions that provide security and confidentiality to our smart contract which in turn bridges the gap between our solution and SePCAR. [48]. As shown in Figure 3.5, when the customer calls the rentCar() function the smart contract verifies two things, the signature of the owner on the booking details and whether it is actually the owner who signed the mutually agreed booking details.(See Code Snippet 3.17).

```
1  const signHash = EthCrypto.hash.keccak256([
2          {
3              type: 'string',
4              value: 'SignedBooking'
5          }, {
6              type: 'address',
7              value: contractAddress
8          }, {
```

```
            type: 'address',
            value: customerIdentity.address
        }
    ]);

    const signature = EthCrypto.sign(ownerIdentity.privateKey,
    signHash);
```

Code Snippet 3.17: Using the owner's private key to sign the customer's address

Initially we make a constant called 'signHash' that stores the result of a hashing procedure on a combination of the contract address, the address of the customer and a hard-coded string, in this case "SignedBooking" which is included to ensure that this signature cannot be replicated in other contracts. This signHash is then signed using the private key of the owner. A similar procedure is done with the booking details, where these details are encrypted as shown in Code Snippet 3.18, and then signed by the owner.

```
function encrypt(publicKey, data) {
    let userPublicKey = new Buffer(publicKey, 'hex');
    let bufferData = new Buffer(data);

    let encryptedData = ecies.encrypt(userPublicKey, bufferData);

    return encryptedData.toString('hex');
}

var enc = encrypt(publicKey,bookDetails);
```

Code Snippet 3.18: Encrypting the booking details using the customer's public key

Now the question arises, "After doing the whole process of signing and encryption off-chain, how do we validate these on the blockchain?". Code Snippet 3.19 shows the procedure to recover the signature done off-chain.

```
    const vrs = EthCrypto.vrs.fromString(signature);
    const vrsDetails = EthCrypto.vrs.fromString(signedDetails);
    const rentCar = contractInstance.methods.rentCar(
    customerIdentity.address, vrs.v, vrs.r, vrs.s, vrsDetails.v,
    vrsDetails.r, vrsDetails.s).encodeABI();
```

Code Snippet 3.19: Retrieval of signature by splitting the signature into three parts — VRS

When the owner signs the 'signHash', it consists of three parts, v (recovery ID), r and s (outputs of an ECDSA signature). The Elliptic curve digital signature algorithm (ECDSA) is widely used in the blockchain technology. We send these three parts separately to the smart contract while calling the rentCar() function where they are used for validation on-chain.

2. The Final Smart Contract:
   Using the JavaScript implementation, we carried out the encryption and the

signing process off-chain. But we also needed to synchronise these functions with our smart contract and validate the signature's made by both parties on-chain.

```
    /**to ensure no replication of signature**/

    string public signPrefix = "SignedBooking";

    //keccak256 is a hash function ethereum uses. prefixHash is
    used to generate a prefixhash of the address

    function prefixHash(address _customer) public constant returns
    (bytes32) {
        bytes32 hash = keccak256(signPrefix,
        address(this), _customer);
        return hash;
    }

    function detailHash(address _customer) public constant returns
    (bytes32) {
        bytes32 hash = keccak256(accessToken,
        address(this), _customer);
        return hash;
    }

    function isSignatureValid(address _customer, uint8 v, bytes32
    r, bytes32 s) view public returns(bool correct) {
        bytes32 mustBeSigned = prefixHash(_customer);
        address signer = ecrecover(mustBeSigned, v, r, s);

        return(signer == carOwner);
    }
```

Code Snippet 3.20: Recovering and validating signature on-chain

Code Snippet 3.20 shows the functions we implemented on the final smart contract to recover and validate owner's signature. We initially make a 'prefixHash' that constitutes of a hard-coded string, and the address of customer and contract and then store this in bytes32 format. After successfully computing the 'prefixHash' we use 'ecrecover' on it to retrieve the signer's address. We then compare if this address is similar to the owner's address, hence verifying owner's signature. By doing this, we ensure Non-Repudiation of the owner's identity and the encrypted booking details (access token).

```
function sendEncryptedDetails(bytes bookingDetails) public{
        accessToken = bookingDetails;
    }
```

Code Snippet 3.21: sending encrypted details to blockchain

Code Snippet 3.21 shows the method that we use to send the encrypted booking details to our local blockchain. Since these details are encrypted using the customer's public key, they can retrieve and decrypt these details off-chain. The decryption procedure is shown in Code Snippet 3.22

```
1
2     function decrypt(privateKey, encryptedData) {
3         let userPrivateKey = new Buffer(privateKey, 'hex');
4         let bufferEncryptedData = new Buffer(encryptedData, 'hex')
      ;
5         let decryptedData = ecies.decrypt(userPrivateKey,
      bufferEncryptedData);
6
7         return decryptedData.toString('utf8');
8     }
9     //retreiving the access token from blockchain and decrypting
      it off-chain.
10    const accessToken = await contractInstance.methods.accessToken
      ().call();
11    console.dir(accessToken);
12
13    const token = accessToken.slice(2);
14    messa = decrypt(privateKey,token)
15
16    console.dir(messa);
```

Code Snippet 3.22: retrieving encrypted details from blockchain and decrypting them using customer's private key

The decrypt function is implemented using JavaScript. The second part of the code snippet explains the procedure to retrieve the encrypted details from the blockchain and using the customer's private key to decrypt them.

3.5 Evaluation

In this section, we will evaluate our smart contract and present the results obtained.

3.5.1 Description of Evaluation Criteria

Our final smart contract can be evaluated using two approaches, namely:

1. The security and privacy properties that it fulfills.
2. The total cost of deploying and using it on the Ethereum blockchain.

3.5.1.1 Security and privacy properties

To bridge the gap between our smart contract based payments and bookings system and SePCAR [48], we will compare its security and privacy requirements and evaluate if our smart contracts fulfills them as follows:

Security Properties

- Confidentiality of booking details: In our smart contract, only the owner and the customer have access to the sensitive information stored in booking details,

as these details are discussed offline and only stored in the smart contract when encrypted.

- Confidentiality of access token: Our smart contract treats the encrypted booking details as the access token and stores them in its internal storage. No one except the customer can decipher this access token.
- Authenticity of booking details: Our smart contracts verifies the owner's signature on the booking details before giving access to the customer.
- Non-repudiation of the origin of access token: Our smart contract verifies the access token and the owner's identity by comparing the address of the signer of the access token to the address of the owner.

Privacy Properties

Our smart contract fulfills the following privacy properties of SePCAR:

- Anonymity of consumer and car: on the blockchain, the consumer can choose to be anonymous by using his 20 byte address to interact with the contract and not publicising any personal details, while the identity of the car is encrypted along with the other sensitive information by the owner before storing it on the smart contract.
- Forensic evidence provision: smart contracts are auditable by nature, and in case of an incident involving the owner or the consumer, our smart contract can be audited to reveal information about the owner or the consumer.

3.5.1.2 Total Deployment and Usage Cost

The deployment of smart contracts and the interaction with these contracts incur a cost to the caller of the transaction, and this cost is calculated in gas. This section discusses the deployment and interaction costs associated with the smart contracts we developed in our case studies. An important thing to mention here is that the price of ether per gas unit is volatile, and these results might not be valid in the future. To evaluate the total cost of our smart contracts, we look at two different kinds of costs, namely deployment cost and transaction costs.

Deployment Cost

The main costs for deploying a contract can be calculated by combining the following [57]:

- The costs associated with storing the contract code (200 gas per byte).
- Additional data storage cost on the contract (20000 gas per 256 bit word).
- Each transaction has a base cost of 21000 gas, and 32000 gas have to be paid for a CREATE transaction (deployment of a new contract).

Table 3.1: Deployment Costs of Smart Contracts Developed in each Case Study.

Case Study	Deployment Cost in gas	Deployment Cost in USD
1	311113 gas	$0.15044
2	437764 gas	$0.2117
3	943016 gas	$0.45603
4	1352283 gas	$0.65395

Table 3.2: Costs of Executing each Transaction in the Final Smart Contract.

Transaction	Cost in gas	Cost in USD
set required days (customer)	42131 gas	$0.02039
store encrypted details in contract (owner)	63022 gas	$0.03047
rent car (customer)	90146 gas	$0.0436
allow use of the car (owner)	28655 gas	$0.01386
access the car (customer)	29038 gas	$0.01402
cancellation of booking (owner/customer)	43035 gas	$0.02079
ending the rental procedure (owner/customer)	82590 gas	$0.03994
triggering the distribution of earnings (owner)	32992 gas	$0.01596
withdrawal of money (owner/consumer)	22099 gas	$0.01068

We used Remix IDE [21] to calculate the deployment costs of our smart contracts, the size of the contract and the total bytes in storage of its constructor influence its absolute deployment costs. As it is illustrated in Table 3.1, the deployment costs of each case study increases as we added more functional requirements and the size of our smart contract increased. The deployment cost in USD can be calculated by multiplying the total gas with the cost per gas unit in USD.

Transaction Cost

The cost for executing each transaction can be calculated as follows [57]:

- The base cost of each transaction (21000 gas).
- The cost for storing a 256 bit word on the smart contract (20000 gas)
- The cost for editing data stored in the smart contract (5000 gas)
- The cost of making a transaction call that contains some value in ether (9000 gas)

Table 3.2 illustrates the cost of executing each transaction in our final smart contract. Based on the results obtained, it can be deduced that our final smart contract is not expensive to deploy and operate.

Chapter 4

Conclusion

4.1 Limitations

Limitations in our smart contract are in the form of security and privacy exploits. In a scenario where the customer makes several bookings for the same car, it could be possible to match the blockchain address with a physical identity. The blockchain address is unique to every user, and whenever the user makes any transaction, he or she has to use the same address. Assuming an adversary could know the owner or the customer, it would be possible to link the transactions of the owner or the customer with their physical identities as all transactions are published on the distributed ledger that is visible to everyone.

Adversaries can also link a user's IP address to his or her account address. If the customer or owner does not use the Ethereum blockchain with Tor [53] or any other encrypted VPN client, it will be possible to link their IP address to an account address, and in this case, all privacy claims would be flawed.

Another thing that our implementation does not cover is the insurance aspect of car bookings. We have implemented countermeasures to control malicious behaviour, but in a scenario where the customer steals the owner's car, or there is an accident, our smart contract does not handle the insurance compensation.

4.2 Discussion

With this thesis, we implemented a booking and payments system that can be incorporated in the work done by Symeonidis et al. [48] that provides a secure and privacy-friendly car sharing protocol called SePCAR. This protocol is a solution to the lack of a secure car sharing community and our implementation aims to contribute to this solution by extending SePCAR's functionalities to enable secure and anonymous booking and payments.

Our final smart contract consists of all major functionalities likely to be present in a full fledged booking and payments platform, it enables an individual to share his underutilised private car with a consumer for personal financial gain along with counter measures to prevent malicious behaviour while being mostly decentralised hence eradicating the need of a Trusted Third Party (TTP).

The booking details of the shared car are initially agreed upon with a mutual consent by the owner of the car and the consumer, most of these details are encrypted off-chain before storing it in the storage of our smart contract to protect the sensitive information about the car and the customer. Since these details are stored on the blockchain, it means that they are immutable, the consumer or the owner cannot change these details. The logic of our smart contract is stored on the Ethereum blockchain, where the contractual clauses are enforced upon our contract's users and the payments are processed in Ether.

Our smart contract fulfills most of the security and privacy requirements of SePCAR, and it can be concluded that smart contracts, presently, are rather cheap and are feasible to be utilised for real world applications such as car sharing systems.

4.3 Future Work

In our future work, we will try to encrpyt all the booking details and still execute the smart contract in a correct manner. In our current implementation, we only encrypt the sensitive booking details that are not necessary for the execution of our smart contract.

Currently we interact with our smart contract using an API written in Javascript. This can also be extended to a full fledged web API that is capable of interacting with our smart contract. It can also enable the owner and the consumer to privately discuss booking details off-chain while having an easy to operate user-interface.

Kosba et al. [32] proposed a system that retains transactional privacy from the public by not storing financial transactions on the blockchain, but this system is still yet to be released. A nice future direction of work would be to use this system to adapt our final smart contract which would enable it to use cryptographic primitives such as zero-knowledge proofs.

Bibliography

[1] S. A. Abbas and A. A. B. Maryoosh. Data security for cloud computing based on elliptic curve integrated encryption scheme (ecies) and modified identity based cryptography (mibc). 2016.

[2] N. Z. Aitzhan and D. Svetinovic. Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams. *In IEEE Transactions on Dependable and Secure Computing*, 2015.

[3] M. Al-Bassam. Scpki: A smart contract-based pki and identity system. *In Proceeding BCC '17, Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, pages 35–40, 2017.

[4] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman. Medrec: Using blockchain for medical data access and permission management. *In 2nd International Conference on Open and Big Data*, pages 25–30, 2016.

[5] J. Balasch, A. Rial, C. Troncoso, B. Preneel, I. Verbauwhede, and C. Geuens. Pretp: Privacy-preserving electronic toll pricing. *In 19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010*, pages 63–78, 2010.

[6] J. H. Bergquist. Blockchain technology and smart contracts. URL: https://uu.diva-portal.org/smash/get/diva2:1107612/FULLTEXT01.pdf, last checked on 2018-07-25.

[7] K. Biswas and V. Muthukkumarasamy. Securing smart cities using blockchain technology. *In IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems, Sydney, NSW, Australia*, pages 1392–1393, 2016.

[8] blockchain.info. Bitcoin hash rate. URL: https://www.blockchain.com/charts/hash-rate, last checked on 2018-07-27.

[9] J. Bouoiyour and R. Selmi. The bitcoin price formation: Beyond the fundamental sources. 2017.

[10] V. Buterin. A next-generation smart contract and decentralized application platform. URL: http://blockchainlab.com/pdf/Ethereum_white_paper-a_

next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf, last checked on 2018-07-20.

[11] V. Buterin. On public and private blockchains. URL: https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/, last checked on 2018-07-20.

[12] E. M. Cepolina and A. Farina. A methodology for planning a new urban car sharing system with fully automated personal vehicles. *In European Transport Research Review*, pages 191–204, 2014.

[13] Coindesk. Bitcoin (usd) price. URL: https://www.coindesk.com/price/, last checked on 2018-07-27.

[14] M. Crosby, Nachiappan, P. Pattanayak, S. Verma, and V. Kalyanaraman. Blockchain technology: Beyond bitcoin. pages 6–8, 2015.

[15] C. David, S. Ponnet, K. D. Wachter, B. Adriaenssen, and M. Thuy. arcade.city: Blueprint for a new economy. URL:https://bravenewcoin.com/assets/Whitepapers/AC-whitepaper.pdf, last checked on 2018-08-1.

[16] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. pages 2–5, 2015.

[17] S. Dhooghe. Applying multiparty computation to car access provision. URL:https://www.esat.kuleuven.be/cosic/publications/thesis-296.pdf, last checked on 2018-08-4.

[18] A. Dmitrienko and C. Plappert. Secure free-floating car sharing for offline cars. *In Proceedings of the seventh ACM on conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdala, AZ, USA, March 22-24 2017*, pages 349–360, 2017.

[19] A. Dmitrienko and C. Plappert. Secure free-floating car sharing for offline cars. *In Proceedings of the seventh ACM on conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdala, AZ, USA, March 22-24 2017*, pages 349–360, 2017.

[20] A. DAlfonso, P. Langer, and Z. Vandelis. The future of cryptocurrency. URL:https://www.economist.com/sites/default/files/the_future_of_cryptocurrency.pdf, last checked on 2018-07-27.

[21] Ethereum. Remix ide. URL:https://remix.ethereum.org/#optimize=true&version=soljson-v0.4.24+commit.e67f0147.js, last checked on 2018-08-4.

[22] Ethereum. solc-js. URL: https://github.com/ethereum/solc-js, last checked on 2018-07-15.

[23] Ethereum. Solidity in depth. URL:http://solidity.readthedocs.io/en/v0.4.24/solidity-in-depth.html, last checked on 2018-08-4.

[24] Ethereum. web3.js. URL: https://web3js.readthedocs.io/en/1.0/, last checked on 2018-07-15.

[25] ethereumjs. ethereumjs-util. URL: https://github.com/ethereumjs/ethereumjs-util, last checked on 2018-07-15.

[26] D. Fireball. Regarding ubers new always location tracking. URL:https://daringfireball.net/2016/12/uber_location_privacy, last checked on 2018-08-4.

[27] D. Gerard. *Attack of the 50 Foot Blockchain: Bitcoin, Blockchain, Ethereum Smart Contracts*. CreateSpace, 2017. ISBN-13: 9781974000067.

[28] J. Giordani. Blockchain – what is it and what is it for? URL: https://www.forbes.com/sites/forbestechcouncil/2018/03/28/blockchain-what-is-it-and-what-is-it-for/#1d2b3df11a16, last checked on 2018-07-23.

[29] T. Guardian. Hell of a ride: even a pr powerhouse couldn't get uber on track. URL:https://www.theguardian.com/technology/2017/apr/14/rachel-whetstone-pr-uber-leave-scandal-crisis, last checked on 2018-08-4.

[30] A. Guzov. Walmart: From supply chain to blockchain. URL: https://rctom.hbs.org/submission/walmart-from-supply-chain-to-blockchain/, last checked on 2018-07-23.

[31] F. Knirsch, A. Unterweger, G. Eibl, and D. Engel. Privacy-preserving smart grid tariff decisions with blockchain-based smart contracts. *In Sustainable Cloud and Energy Services*, pages 85–116, 2017.

[32] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. *2016 IEEE Symposium on Security and Privacy (SP), 22-26 May 2016, San Jose, CA, USA*, pages 839–858, 2016.

[33] La'Zooz. La'zooz white paper. URL:https://www.weusecoins.com/assets/pdf/library/LaZooz%20Blockchain%20Taxi%20Whitepaper.pdf, last checked on 2018-08-1.

[34] J. Lee, J. Nah, Y. Park, and V. Sugumaran. Electric car sharing service using mobile technology. *In Confirm Proceedings, paper 12*, 2011.

[35] libertylocked. eth-ecies. URL: https://github.com/libertylocked/eth-ecies, last checked on 2018-07-15.

[36] V. G. Martinez, L. H. Encinas, and A. Q. Dios. Security and practical considerations when implementing the elliptic curve integrated encryption scheme. *Cryptologia*, 39(3):pages 244–269, 2015.

[37] V. G. Martinez, L. H. Encinas, and C. S. vila. A survey of the elliptic curve integrated encryption scheme. *Journal of Computer Science AND Engineering*, 2:pages 7–13, 2010.

[38] M. A. Mustafa, N. Zhang, G. Kalogridis, and Z. Fan. Roaming electric vehicle charging and billing: an anonymous multi-user protocol. *In IEEE SmartGridComm*, pages 939–945, 2014.

[39] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. URL: https://bitcoin.org/bitcoin.pdf, last checked on 2018-01-20.

[40] K. J. O'Dwyer and D. Malone. Bitcoin mining and its energy footprint. *In 25th IET Irish Signals Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014), Limerick, Ireland, 26-27 June 2013*, 2013.

[41] pubkey. eth-crypto. URL: https://github.com/pubkey/eth-crypto, last checked on 2018-07-15.

[42] P. Rogaway and M. Bellare. Minimizing the use of random oracles in authenticated encryption schemes. *Lecture Notes in Computer Science*, 1334:pages 1–16, 1997.

[43] P. Rogaway, M. Bellare, and M. Abdalla. Dhies: An encryption scheme based on the diffie-hellman problem. *Cryptology ePrint Archive, Report 1999/007*, 1998.

[44] P. Rogaway, M. Bellare, and M. Abdalla. The oracle diffie-hellman assumptions and an analysis of dhies. *Lecture Notes in Computer Science*, 2020:pages 143–158, 2001.

[45] P. Ruiz. A smart contract for a smart car. URL: https://hackernoon.com/a-smart-contract-for-a-smart-car-db08eda4bb4f, last checked on 2018-01-20.

[46] F. Schuepfer. Design and implementation of a smart contract application. URL:https://files.ifi.uzh.ch/stiller/Thesis-F-Schuepfer-final.pdf, last checked on 2018-08-4.

[47] Solidity. Contract abi specification. URL: https://solidity.readthedocs.io/en/develop/abi-spec.html#, last checked on 2018-07-23.

[48] I. Symeonidis, A. Aly, M. A. Mustafa, B. Mennink, S. Dhooghe, and B. Preneel. Sepcar: A secure and privacy-enhancing protocol for car access provision. *in the 22nd European Symposium on Research in Computer Security (ESORICS17), ser. LNCS, vol. 10493. Springer, 2017*, pages 475–493, 2017.

[49] I. Symeonidis, M. A. Mustafa, and B. Preneel. Keyless car sharing system: A security and privacy analysis. *In IEEE International Smart Cities Conference (ISC2 2016)*, pages 1–8, 2016.

[50] N. Szabo. Smart contracts. URL:http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html, last checked on 2018-07-27.

[51] D. Tapscott and A. Tapscott. *Blockchain Revolution: How the Technology Behind Bitcoin Is Changing Money, Business, and the World*. Portfolio, 2016. ISBN-13: 978-1101980132.

[52] Tinny. A beginner's guide to the different types of blockchain networks. URL: https://blog.xsolus.com/different-types-of-blockchain-networks, last checked on 2018-07-20.

[53] Tor. Tor browser. URL:https://www.torproject.org/projects/torbrowser.html.en, last checked on 2018-08-4.

[54] C. Troncoso, G. Danezis, E. Kosta, J. Balasch, and B. Preneel. Pripayd: Privacy-friendly pay-as-you-drive insurance. *IEEE Transactions on Dependable and Secure Computing*, 8(5):pages 742–755, 2011.

[55] trufflesuite. ganache-core. URL: https://github.com/trufflesuite/ganache-core, last checked on 2018-07-15.

[56] Wikipedia. Blockchain. URL: https://en.wikipedia.org/wiki/Blockchain, last checked on 2018-07-20.

[57] G. Wood. Ethereum: A secure decentralised generalised transaction ledger byzantium version. URL:https://ethereum.github.io/yellowpaper/paper.pdf, last checked on 2018-08-5.

[58] G. Zyskind, O. Nathan, and A. S. Pentland. Decentralizing privacy: Using blockchain to protect personal data. *In Security and Privacy Workshops (SPW), 2015 IEEE*, pages 180–184, 2015.

Master's thesis filing card

Student: Akash Madhusudan

Title: Applying Smart Contracts to Secure Car Sharing Systems

Dutch title: Het toepassen van Smart Contracts op beveildigde systemen voor autodelen

UDC: 681.3*I20

Abstract:

The appearance of Consumer-to-Consumer (C2C) markets has introduced a possibility for individuals to share their personal underutilised assets for a short- or long-term. However, their reliance on a trusted third party leads to various privacy concerns. Car sharing systems such as car2go or Zipcar are gaining popularity, and the use of in-vehicle telematics has introduced systems that allow users to share their cars conveniently without requiring traditional keys. These systems are called key-less car sharing systems. Although such systems offer convenience to users, they also introduce several security and privacy challenges.

A solution that addresses some of these challenges and allows individuals to share their personal car in a secure and privacy-friendly way is SePCAR, a protocol for car access provision proposed by Symeonidis et al. [48]. This thesis is about designing a decentralised booking and payments platform for SePCAR so as to eradicate these security and privacy challenges and allow these car sharing systems to operate without a trusted intermediary. Our implementation makes the use of smart contracts deployed on the Ethereum blockchain that provides full-fledged car sharing functionalities along with various countermeasures to tackle malicious behaviour.

Thesis submitted for the degree of Master of Science in Artificial Intelligence, option Big Data Analytics

Thesis supervisor: Prof. dr. ir. Bart Preneel

Assessors: Prof. dr. ir. Frederik Vercauteren
Andreas Put

Mentors: Dr. Mustafa A. Mustafa
Dr. Iraklis Symeonidis