A Peer-to-Peer Purchase and Rental Smart Contract-based Application

Sina Rafati Niya, Florian Schüpfer, Thomas Bocek, Burkhard Stiller

Abstract:
Many applications in various groups of use cases employ unique features of blockchains (BC) as distributed and decentralized ledgers and Smart Contracts (SC) as enablers of transactions in BCs. This work introduces the design and implementation of an Android-based Peer-to-peer (P2P) purchase and rental application, which leverages SC and Ethereum public BC. This application allows users to flexibly specify purchase and rental contracts, to exchange personal user data and, to execute monetary transactions directly on the BC.Thus, all payments are conducted in Ethereum currency Ether. As a Device-to-device (D2D) communication protocol, WiFi-Direct is chosen to enable the P2P data transmission between two parties due to the capability of transferring high data rates in comparison with other protocols. As an addition to the basic trade contract introduced in Ethereum, SCs in this application are developed in a way to use deposits to increase compliance trust between parties in a contract. This work results in a cost-efficient, legally valid, secure, SC-based, P2P, and Decentralized application (Dapp). Relatively, Evaluations on performance of this Dapp is specified in terms of its D2D deployment, transaction costs, scalability, security, and privacy.

ACM CCS: Computer Systems Organization→ Architectures → Distributed Architectures → Peer-to-peer Architectures→ Blockchains

Keywords: Smart Contracts, Blockchains, Ethereum, Peer-to-peer Applications, Decentralized Mobile Applications

1 Introduction

In the last decade, the Business-to-Consumer (B2C) platforms dominated e-commerce business. With the introduction of the first Consumer-to-Consumer (C2C) platforms, a new form of markets emerged where private persons can come together and exchange goods and services directly [29]. This market has developed from simple sale and resale platforms like Ebay [18] to more advanced short- and long term rental platforms. Examples for such platforms are Airbnb [1], where private persons can rent out their unused rooms to other persons or RelayRides [36], where people can share their unused vehicles. The phenomenon of the rapid growth of the C2C market in the last years is often referred to as the "Sharing Economy" [29].

A common problem that all these platforms share is that they rely on a Trusted Third Party (TTP), the platform owner, to operate the platform. This has many disadvantages for consumers. They need to register at each platform separately and typically have to give away their private data to the platform owner. They also often have to pay transaction fees [6]. A TTP also determines a single point of failure for all participants of the platform. It can be attacked or it can be shut down because the platform owner has to comply with a central authority [34].

A solution to overcome the requirement of a TTP in a contractual agreement between two or more parties is offered by the concept of Smart Contracts (SC). The concept of a SC was first introduced by Nick Szabo [40]. He defined it as a digital protocol that facilitates the agreement process between different parties by enforcing certain predefined rules. [40] proposes to embed contractual clauses, like property rights directly into hard- and software to make it expensive for a party to breach the protocol. In [40] a hypothetical digital security system for cars is outlined, where a cryptographic key repres-

ents the ownership of the car and is transferred only if the terms implemented in the protocol are fulfilled. In his example, the owner of a car can withdraw the key from a leaser who does not pay the monthly rent.

At that time, SC were not practically achievable, mainly because no digital infrastructure existed that allowed the secure execution of protocols without a TTP. This changed with the introduction of the Crypto-Currency (CC) Bitcoin and its underlying blockchain (BC) technology.

Bitcoin [33] defines a CC for transfer of funds between participants without relying on a TTP, like a bank. The ownership of currency tokens is maintained with the help of public key cryptography, meaning that a token belongs to the user that holds its associated private key. To maintain the integrity of the network, Bitcoin introduced the BC, a distributed ledger that stores all transactions ever committed in a chain of distinct blocks. Participants have to solve a cryptographic puzzle that is used to verify a block. The main incentive for participants (miners) to take part in this process is a reward that is distributed when a block is proofed to be valid. In this manner, new Bitcoin tokens are created and this processes is called mining.

Although the functionality of Bitcoin can be extended beyond the exchange of value tokens by writing special protocols, the scripting language used is not powerful enough to express complex contract logic like in the example envisioned by [16].

Ethereum [20], a cryptocurrency described first by [11], is the first cryptocurrency with full support for SC. It stores the code and related data of a contract on the BC and executes code when participants or other contracts issue transactions. The code of a contract is stored in a byte code format and executed concurrently by all participants on the Ethereum Virtual Machine (EVM). Ethereum supports several scripting languages to write SC. The most popular among them is Solidity, a contract-oriented high level language whose syntax is similar to JavaScript (JS). Various types of apps can be implemented in Solidity, like “Voting”, “Crowd Funding”, “Blind Auctions” or “Multi Signature Wallets” [23], [20].

Public BC of Ethereum provides a P2P and fully decentralized architecture, which by enabling and adoption of SC there is a potential in solving TTP issues in developing Dapps using this BC. There are various challenges to be addressed in design and implementation of a Dapp with such a use case, such as choosing the best protocol for D2D communications, providing trust, privacy, and security on one hand, and on the other hand, developing the SC codes to be as cost efficient as possible with high scalability and ability of handling and managing transactions for setting up successful contracts.

In this paper a Dapp will be presented which is designed and developed to tackle the issues experienced with traditional contracts. These problems can be listed as: (1) Need of a TTP to verify transactions, (2) Need of a monetary organization like banks to store and manage users money for online transactions, (3) Need of paper work for setting up contracts. Proposed Dapp is designed and implemented to leverage the decentralized nature of BCs, and specifically Ethereum BC, to tackle the centralized TTP requirement problems in purchase and rental contracts. This Dapp is developed for Android-based mobile phones with a P2P architecture.

One challenge in designing apps to be equipped with D2D communication between the peers is to provide/choose the potentially best method/protocols for D2D communications. According to the required throughput and security of such communication the WiFi-Direct protocol is used to exchange identity information between users. Users are encouraged to use either their phone camera to scan a QR code or by exchanging it through WiFi-Direct. Comparisons of D2D protocols, is mentioned in section4.

In order to increase the scope of applicability, legal aspects of SC are investigated in this article to address the question if it is possible to conclude a legally valid electronic purchase or rental contract directly between two private parties without the need of a TTP. According to results of these investigations, this Dapp allows users to flexibly set up contracts that provide enough information to be legally valid. To provide security and privacy, in order to exchanging sensitive user data between two parties for identification purposes, user data is encrypted during transmission and signed before sending over the network.

One of the most important aspects in developing apps using SC, is the cost for saving transactions in the BC. To tackle high costs of storing data, this paper introduces light weight contracts along with full contracts. Additionally, data types are carefully regarding units and functions in the SC to obtain the least contract cost. Also, as users can decide themselves the amount of identity-related data to be provided for each contracts by both sides, users are capable of reducing total costs by minimizing the requested data. to calculate SC costs. Estimation equations are developed to calculate SC costs in advance with a high accuracy.

This paper is organized as follows, section 2 reviews related work. Legal aspects of purchase contracts are described in Section 3 and legal aspects of SCs are explained in Section 3.1. A brief overview of WiFi-Direct communication protocol is provided in Section 4. Design decisions are presented in Section 5 and implementation of the whole system including SC codes and D2D connections are described in detail in Section 6. Section 7 is starts with cost evaluations of the implemented SC codes including the empirical results and then, the analysis of scalability, privacy, and security are performed in Sections 7.2, 7.3, and 7.4 respectively. Final concluding remarks are mentioned in Section 8.

2 Related Work

By looking in the area of P2P apps, it's clearly obvious that there are not a lot of Dapps to employ BCs and SC to develop mobile apps with the purpose of flexible (in term of information from both parties to be presented to each other and accessing other parties information by D2D communications), legally valid and safe transaction execution (guaranteed purchasing and renting transactions) for the Purchase ans renting use case. In this section an overview of existing apps are introduced.

An app in [6] presented which, uses SC and Ethereum for the conclusion of rental contracts. In this app, the creator of a contract can register rental objects that are stored in a dictionary and referenced by their Id's. SC stores the description of the item together with its rental price and deposit. When one party wants to rent an object, she scans its QR code with the camera of her smart phone. The client then invokes the *rentObject* function on the contract and submits the deposit for the object. The renter is then assigned to the object together with a timestamp. When the object is returned, the owner of the contract can then trigger the *reclaimObject* function that calculates the renting fees for the time of usage and sends the renting fee to the address of the owner. The deposit is then returned to the renter. Front end of the app that provides the user interface is implemented in JS and HTML5.

The most important difference between the introduced dapp and the work of bogner et all, is that the decentralized sharing app uses a central SC to manage existing rent items and is more focused on the commercial renting of items from a store. However, this dapp focuses more on direct unique transactions between two parties and also supports purchase contracts. Another difference is that the introduced dapp supports the exchange of personal information between participants. Since it is an Android app, the client does not rely on a web server and has the potential to be completely independent of any centralized infrastructure when mobile Ethereum clients are available in the future.

TransActive Grid [41] is a P2P energy trading platform that uses Ethereum based SC to manage transactions between participants in a local electric power microgrid. In the system, every owner of a "photovoltaic"facility installs a smart meter that keeps track of the surpluses she makes. The smart meter then updates the available surplus for this participant on a SC on the Ethereum BC. Interested buyers that live in the same neighbourhood can then interact with the contract to buy energy credits. Brooklyn microgrid [7] is the first renewable energy startup that uses the Transactive Grid system to create a local energy market. This system is not yet fully decentralized because it relies on the payment platform Paypal to conduct the financial transactions.

Digix [17] is an asset tokenisation service built on the Ethereum BC. Digix is offering physical gold bullions on the BC, more specifically, digital tokens linked to real world physical assets. DGX tokens are created ("minted") through an Ethereum SC with each token representing 1 gram of gold. Each DGX token can be linked definitely to a Proof of Asset (PoA) card of a physical gold bar. The PoA card is stored on a SC on the BC and contains information like the time stamp of the card creation, the gold bar serial number, the purchase receipt and the audit documentation [13]. The PoA asset card is created through the "Asset Registration Process". The registration process accepts a gold contract from a user and creates the PoA card that is linked to the Ethereum address of the user. To convert the PoA asset card to DGX tokens, a user can use the "Minter SC"that will hold the PoA card and return 1 DGX token per gram of gold. With the "Recaster SC"a user can exchange its DGX tokens back into PoA cards. To redeem the PoA card back to the physical gold bar, the user can trigger the "Redemption Process"[13].

Main benefit of DGX tokens is that their value is much more stable than the cryptocurrency *Ether* (ETH), because they are directly backed by gold or other physical assets. Another benefit of DGX compared to traditional digital gold certificates is that this system does not rely on a centralized database that holds the information stored in a PoA card. Further, This app is prone to "Man In The Middle "(MITM) attacks because users use a desktop client to log in, instead of a web form [13].

A comparison between the related apps are listed in table1. In this table, SC Dapp is presented in this article which, regarding the BC used it is the same as other apps but considering the use case and currency, there is only the Bogner Dapp is the closest one which has the renting functionality and uses *Ether*. Considering the client platforms, Transaction Grid and Digix are developed for desktop machines, while Bogner and SC Dapps are specifically designed for Mobile phones and considering decentralization, SC Dapp is capable of being fully decentralized by employing Ethereum light clients on the smart phones.

Table 1: Comparison of related apps

	BC technology used	**app Use-Case**	**Payment method**	**Client Platform**	**Fully decentralized**
Bogner Dapp	Ethereum	P2P rental contracts	Ether	Mobile	No
Transactive Grid	Ethereum	P2P energy trading	Paypal	Desktop	No
Digix	Ethereum	Physical asset tokenization	Gold and other metals	Desktop	No
SC Dapp	Ethereum	P2P rental and purchase contracts	Ether	Mobile	Yes

3 Legal Aspects Purchase and Rental Contracts

This section will focus on the legal aspects of purchase and rental contracts and the requirements need to be met to provide users legally bonding contracts. A purchase contract should not only be legal (accepted by law), but also be enforceable by law. This poses the question about the required properties a purchase contract must have to offer legal security.

Law doesn't give a definition of a purchase contract but describes duties of involved parties that result out of conclusion of a purchase contract: It obligates the seller to hand out the purchase item and to transfer the property of it to the buyer and it obligates the buyer to pay the purchase price to the seller [10]. Buyer has to pay the purchase price to seller. He also must accept the item that is offered to him in the purchase contract. If nothing else is negotiated or commonly accepted, reception of an item must take place immediately. Further, buyer has to review the purchase item after reception and notify the seller about possible defects [41].

Conclusion of a purchase contract needs general rules on contract conclusions which in this case we consider the swiss federal law defined in *OR 1 ff* [39] have to be applied. The most important rules are briefly described here:

- **Conformable will:** It is crucial that both parties express their will to conclude the contract. It is also mentioned that this declaration can be explicit or by implication. This means that the relationship between the parties may be created orally, in writing or by conduct. For example, when a party accepts a good or a service against payment, this is considered a declaration of intent by conduct [9]. However, an offer in an online-shop is usually not considered an offer in legal terms, but an offer to the customer to send a bid to the seller. This means that the offer is made when a customer orders a good or a service. The seller has to accept this offer for a contract to be formed. He can accept this offer either explicitly or by implication by sending the goods to the customer [35]. Alternatively, [9] says that the applicant of an offer is not legally bound to it if he adds a disclaimer of warranty to the offer or if such a caveat lies in nature or in the circumstances of an offer [38].
- **Requirements for Validity:** The purchase contract has to fulfill specific formal requirements for special kinds of contracts (for example when buying real estate). This is discussed in more detail in section 4.3. Further the purchase contract has to conform to the barriers of contractual freedom. This means that the content of the contract can be chosen freely unless it violates the law or it is an agreement against public policy [41].
- **Articles of Agreement:** For the conclusion of a contract the parties need to agree at least on the essential articles of agreement *(Essentialia Negotii)*. For the purchase contract, both parties must agree on the price and the purchase item for the contract to be valid [10].

3.1 Legal Validity of SC-based Applications

One of the first issues come to mind about SC-based apps while comparing to regular contracts is whether these contracts are legally valid or they need to adopt some specific the Its crucial to deduce the legal binding contracts, which are produced and used by SC-based app. Nature of offers in SC are very different from an offer in an online shop. In a conventional online shop, sellers does not have to accept an offer from customers and buyers only have to pay the purchase price if sellers confirmed the purchase. In contrast, SC code allows only one buyer per offer and account of the buyer is immediately charged after clicking on the *"Buy"* button. Further, SC is locked after the payment of the buyer is confirmed and neither party can withdraw the money any more.

It can be argued that such an offer is binding by nature because the SC code does not allow the withdrawal of payment after an offer is accepted by the buyer. To make sure that an offer made by seller is binding, it is better to specify this explicitly in the general terms and conditions of the app. For example, general terms and conditions of "Ebay"also specify these conditions for offers made by their customers [19].

Articles of an agreement are fulfilled if the price and the purchase item are defined properly. Since the SC requires the seller to specify the price in a common currency, the first condition is satisfied. Regarding the definition of the purchase item, the law does not specify in which form and how detailed this description must be. Therefore, it's been assumed that a written description of a purchase item and its properties together with one or more pictures of it are sufficient.

Regarding the formal requirements, it can be stated that every SC is valid as long as its content does not require the contract to fulfill special formal properties. This means that for example contracts involving the purchase of real estate, patents, designs or trade markets cannot be concluded with the SC.

Considering the above mentioned requirements, implemented app is legally valid as the following conditions are fulfilled: (1) General terms and conditions explicitly state that an offer made by the seller is binding by law. (2) Content of the SC does not violate the law and is not an agreement against public policy. (3) The content of the SC does not have special formal requirements by law.

4 Device-to-device Communication

In order to provide the means of sending and receiving the identity data, there are few protocols to choose such as Near Field Communication (NFC), WiFi, WiFi-Direct, Bluetooth and Bluetooth Low Energy (BLE). A brief overview of D2D communication protocols are listed in Table 2. Different wireless technologies may differ in operating frequency range, data rate and security standards. Obviously, selecting the best protocol depends on the requirements and use cases, which are high data rate and security in the introduces SC-based Dapp.This section describes and justifies the incentives of using WiFi-Direct protocol as the D2D communication protocol in the presented Dapp.

WiFi-Direct is an extension of the IEEE 802.11 protocol and was developed for direct communication of two or more devices in the absence of a distribution network.

Table 2: Overview of wireless technologies in D2D communications

	Bluetooth	**BLE 4.0**	**WiFi-Direct**	**NFC**
IEEE Standard	802.15.1	802.15.1	802.11 (a, b, g, n)	ISO 18092
Frequency (GHz)	2.4	2.4	2.4 and 5.0	0.01356
Max. Data Rate (Mbps)	1-3 (24 with HS)	1	11 (b), 54(g), 600 (n)	0.106, 0.212 or 0.424
Security	128 bit SAFER+	128 bit AES, user defined on app layer	256 bits AES-CCMP	short range, user defined on app layer

WiFi-Direct uses the WiFi Protected Setup (WPS) procedure to secure the connection with minimal user intervention [12]. WPS is based on the WiFi Protected Access 2 (WPA2) protocol. WPA2 implements the IEEE 802.11i standard and provides data confidentiality and integrity by using the Advanced Encryption Standard (AES)-CCMP cypher. When used in Personal mode, a Pre-Shared-Key (PSK) must be present both at the AP and the client for the mutual authentication. The 256 bit PSK is usually generated from a plain text pass phrase that must be entered on both devices. After the authentication, a set of temporary keys is exchanged between the AP and the client which are regularly updated [3].

WPA2 is considered secure against most attacks like man-in-the-middle (MITM), authentication forging, replay, key collision, weak keys, packet forging and brute-force attacks [3]. Using WPS as authentication mechanism, the PSK can be exchanged with less user intervention than usually required by using one of the following methods:

- **Push-Button-Connect(PBC):** The user has to press a button on both the AP and the client device [42].
- **PIN:** The user has to enter the PIN of the WiFi adapter into the web interface of the AP (Internal registrar) or the user has to enter the PIN of the AP in a UI form on the client device (External registrar) [42].

It has been shown that the External Registrar authentication is potentially vulnerable to brute-force attacks, because the authentication is entirely based on the PIN [42]. WiFi-Direct uses the PBC authentication method, which is vulnerable against man-in-the middle"(MITM) attacks during the authentication phase, which is only active when the buttons are pressed on both devices [28],[42]. Considering throughput, since both the user profile and the contract can contain high resolution images, the payload can easily exceed 1 megabyte. Both classic Bluetooth and Bluetooth Low Energy do not provide the bandwidth to transmit larger files in reasonable time, especially since the achievable data rates will be below the theoretical limit. Bluetooth 3.0+HS could provide up to 24 Mbps, but is not very widely used among smart phone users as well as NFC which is most of the times users don't know how to run the NFC or they can't find the exact location of the NFC hardware in their phones. Therefore only WiFi-Direct can provide the required throughput.

And Considering security issues, The user profile contains personal data and therefore the transmission should be encrypted appropriately. WiFi-Direct has a stronger encryption by using 256 bit key length but it is also vulnerable against man-in-the-middle (MITM) attacks during the WPS authentication procedure. Both Bluetooth and BLE offer OOB and PIN authentication to exchange keys and can be considered safer when these mechanisms are used.

Although the exchanged data is personal and allows a potential adversary to identify the parties, it does not provide her with information that could be used directly to steal money out of a contract or to use the identity of a party to sign a contract. Even if the parties could sign a contract with a certified signature in the future, the private key would never be transmitted to the other device. Therefore, security is certainly important but does not have the same priority than data throughput in this use case.

5 Design

The proposed Dapp's architecture is designed as a P2P system in which users hold Android-based smart phones and at the same time, they are connected to Ethereum clients. Design of this Dapp is based on two main methodologies. In First method, users will install Ethereum light client on their smart phones and thus, they are connected directly to the Ethereum BC as shown in right side of the fig 1. In second method, users need to connect to an Ethereum client which may be installed

on a laptop machine as shown in the left side of the fig 1.

Identity provision is also designed to be possible in two methods. First method is using QR readers in which each party can read the other parties identity by reading his/her QR image. Second method is to send and receive identity data via WiFi-Direct with possibility of setting the required ID data by seller or hirer flexibly, e.g., asking for buyer's image in addition to name and email, etc., as shown in fig 3-c. The latter design decision made to provide higher privacy for users as this option will facilitate this Dapp to provide as less identity information of users to other sides as possible even though they have already entered all those information (e.g., 1:QR code and 2: image) in his profile as shown in fig 2.

SC are designed to establish trust for both sides of contracts by asking them to deposit the same amount of money in terms of *Ether* to avoid any malicious behaviour. Details of SC implementation are presented in following sections.

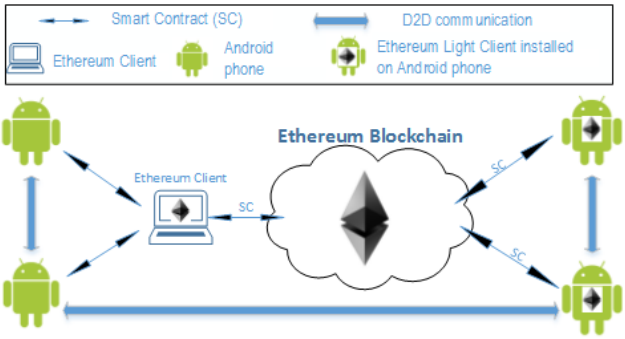

Figure 1: Overview of the Architecture Design

6 Implementation

In this P2P system, Android clients use service interfaces of the Java library to load and deploy SC. The Java Web3j library is used to wrap the interface of a SC in a Java class which is then used by the Android client to execute transactions on the BC. Web3j uses the JSON-RPC interface of an external Ethereum client that is connected to the Ethereum network to deploy contracts and interact with them as shown in fig 5. SC codes are developed with Solidity language and their compiled byte-code is stored in the associated Java wrapper class.

6.1 Rental and Purchase Smart Contracts

This section discusses the Solidity code for the developed purchase and rental SCs in this Dapp. Users can setup purchase or rental contracts with the Android client by

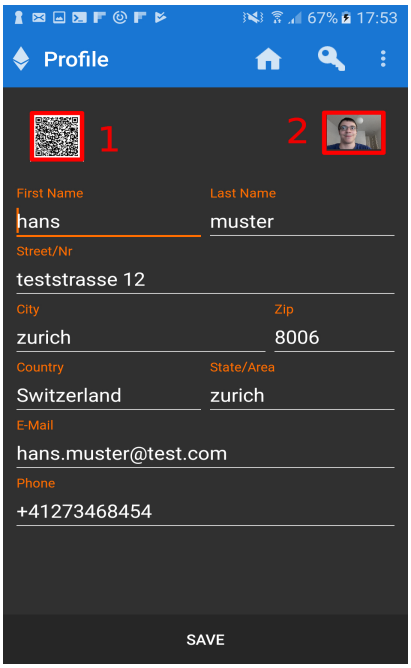

Figure 2: User profile

creating new contracts or, using already existing ones as shown in fig 4. Listing 1.1 shows the base contract used for both contracts that stores details of the rented or purchased items and of the parties.

In the purchase SC, to conduct a safe purchase both seller and buyer have to pay a deposit before transaction can take place and deposited assets are only released when the buyer confirms reception of the item.

In the rental SC, on one side hirer can specify renting and deposit fees for an item. On the other side, renter has to pay the deposit beforehand and only gets it back after he returned the item and paid the renting fee.

Listing 1.1: The TradeContract

```
1  pragma solidity ^0.4.8;
2  contract TradeContract{
3  string public title;
4  string public description;
5  uint public price;
6  uint public deposit;
7  address public seller;
8  address public buyer;
9  bool public verifyIdentity;
10 bytes32[] private imageSignatures;
11 enum State { Created, Locked, Inactive,
     AwaitPayment }
12 State public state;
13 function TradeContract(string _title, string
     _description,bool _verify, uint _deposit,
     uint _price, bytes32[] _imageSignatures)
     {...}
14 modifier require(bool _condition) {
15  if (!_condition) throw; _ ;}
16 modifier onlyBuyer() {
17  if (msg.sender != buyer) throw; _ ; }
19 modifier onlySeller() {
20  if (msg.sender != seller) throw;_ ; }
21 modifier inState(State _state) {
22  if (state != _state) throw; _ ; }
23 event aborted();
24 function abort();}
```

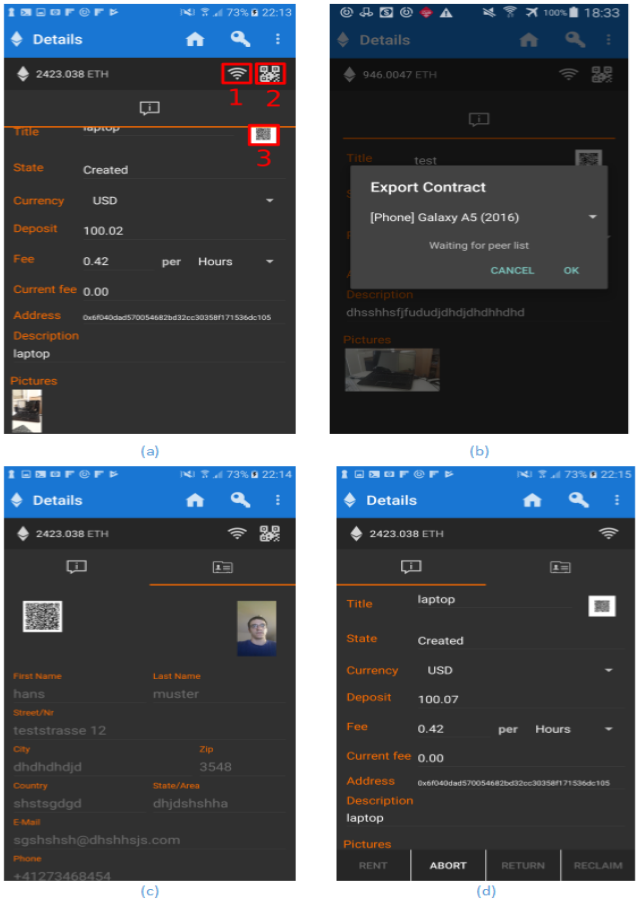

Figure 3: Process of purchasing a contract, (a) Information of a contract before setting up, (b) Setting up a contract, (c) Buyer identity information, and (d) Created contract's information

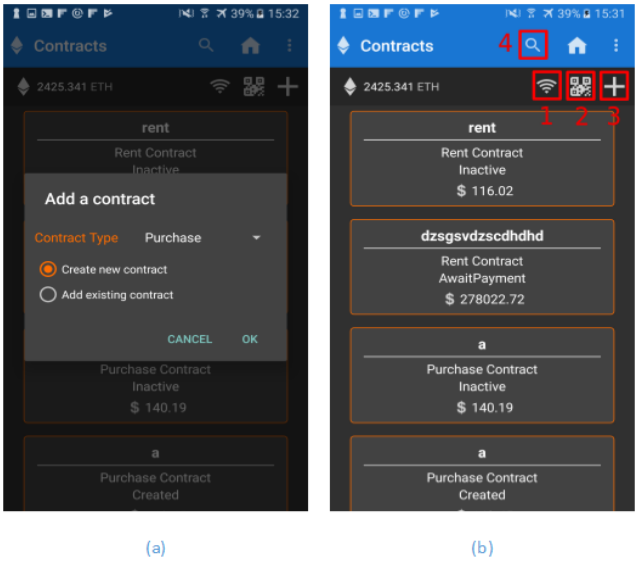

Figure 4: UI for Setting up new contracts or accessing previous contracts

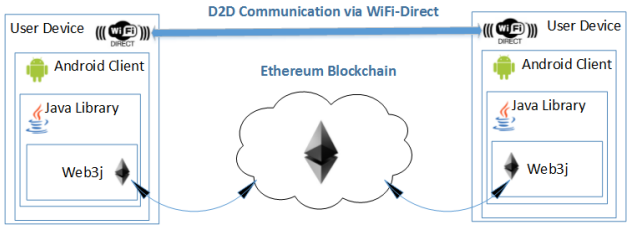

Figure 5: Overview of implemented Android and Ethereum clients integration architecture

In each contract, description of an item is composed of a title, textual description and an array of image signatures as shown in fig 3. An image signature is calculated by using the SHA256 cryptographic hash function of an image taken by the Android client. The reason for not storing a whole image in the Ethereum BC is the cost associated with the high storage usage. Assuming that a PNG or JPG image with reasonable resolution has a size of 300 kB and cost of storing one 256 bit word on the Ethereum BC is 20000 units of gas, cost for storing one image can be calculated by:

$$P_{storage} = 937.5 \times 20000 \times P_{gas} \times P_{wei} \qquad (1)$$

Where *Pstorage* is the price for storing the image in USD, *Pgas* is the gas price and *Pwei* is the dollar exchange rate for 1 *wei*. Absolute price can vary because both gas price and exchange rate for *Ether* have a high volatility. As of July 16, 2017, the average gas price is 4 *gwei* [24] and the dollar exchange rate for 1 *Ether* is approximately 175 $ [10]. This would result in a price of 131.25 $ for storing one single image!

Every contract also stores *address* of the 2 contractual partners as well as *price* and *deposit* for the item. The *deposit* is used to ensure contractual compliance of the parties. Finally, contract stores a boolean value that indicates whether the parties should exchange personal information. This indicator is used by client app to determine whether personal information like addresses, photos or other data should be exchanged between the parties. Every contract can be aborted in certain circumstances and therefore declares the aborted function and an aborted event.

6.1.1 Purchase Smart Contract

Purchase contract is derived from the *TradeContract* and it is a modified version of the purchase contract presented in the official Solidity documentation [23] as shown in listing 1.2.

Listing 1.2: The Purchase contract

```
1  contract Purchase is TradeContract{
2  function Purchase(string _title, string
     _description, bool _verify, bytes32[]
     _imageSignatures) payable
3  TradeContract(_title, _description, _verify,
     msg.value / 2, msg.value / 2,
     _imageSignatures){
4    if (2 * price != msg.value) throw;}
5  event purchaseConfirmed();
6  event itemReceived();
7  function abort()
```

```
8  onlySeller
9  inState(State.Created){
10 aborted();
11 state = State.Inactive;
12 if (!seller.send(this.balance)) throw;}
13 function confirmPurchase()
14 inState(State.Created)
15 require(msg.value == 2 * price)
16 payable{
17 purchaseConfirmed();
18 buyer = msg.sender;
19 state = State.Locked;}
20 function confirmReceived()
21 onlyBuyer
22 inState(State.Locked){
23 itemReceived();
24 state = State.Inactive;
25 if (!buyer.send(deposit) ||  !seller.send(
    this.balance)) throw;}}
```

To give an incentive to a seller to not betray a buyer, the seller has to transmit *Ether* in the constructor of the contract. As can be seen in listing 1.2 lines 3-5, the value provided must be dividable by 2, otherwise the contract throws an exception. Thus the actual price of the item is only half the value provided in the constructor. When the buyer executes the *confirmPurchase* function, she also has to pay a deposit together with the actual purchase price. The state of the contract is then set to "Locked". From this point on, the funds of the 2 parties are locked and can only be released when the buyer confirms that she received the item. Locking of deposits makes sure that no party has a monetary advantage in betraying other side since both have invested the same amount of money in the contract and non of them can get their deposit back when the other party does not comply.

As long as the contract is in the state *Created* meaning that buyer has not accepted the offer, seller can execute "abort"function and the balance stored in the contract is refunded to him. In the "confirmReceived"function the buyer confirms that she received the item. The state of the contract is set to Inactive meaning that no further interaction with it is possible. The deposit is refunded to the buyer and the rest of the balance stored in the contract is sent back to the seller. Notice that the state is changed before the *Ether* is transmitted with the send function. This prevents an attacker from calling the function again from its fallback function. Every time the state of the contract changes, an event is emitted on the event log. This allows the client app to update its UI and inform the buyer or seller that the state has changed.

6.1.2 Renting Contract

In the renting contract as presented in listing 1.3, buyer assumes the role of a renter and the seller assumed as the role of a hirer. The hirer defines details of the rent item together with its renting price and deposit in constructor of the contract. Renting price is provided in *wei* per second. As long as the contract is in the *Created* state, hirer can execute the abort function. When the renter executes the rent function, she has to pay the deposit for the item. State of the contract is set to Locked and start time for the contract is initialized. When the renter wants to return an item, the hirer executes the *reclaimItem* function which calculates the renting fee based on the renting time and the price and sets the state of the contract to *AwaitPayment*. In the *AwaitPayment* state, renter can execute *returnItem* function, which then returns the deposit and the change to the buyer. The renting price is sent to the renter. In the *AwaitPayment* state, the hirer can also execute the *reclaimItem* function again in the case the renter does not pay in time.

Listing 1.3: The Rental contract

```
1  contract Renting is TradeContract{
2  uint256 private rentingFee;
3  uint256 private start;
4  function Renting(string _title, string
    _description, bool _verify, bytes32[]
    _imageSignatures, uint _deposit, uint
    _rentingPrice)
5  TradeContract(_title, _description, _verify,
    _deposit, _rentingPrice, _imageSignatures)
    {}
6  event itemRented();
7  event itemReturned();
8  event paymentRequested();
9  function abort()
10  onlySeller
11  inState(State.Created){
12    aborted();
13    state = State.Inactive;}
14  function rentItem()
15  inState(State.Created)
16  require(msg.value == deposit)
17  payable{
18    start = now;
19    itemRented();
20    buyer = msg.sender;
21    state = State.Locked;}
22  function calculateRentingFee()
23  returns(uint256){
24    if(state == State.Inactive)
25        return rentingFee;
26    if(start == 0)
27        return 0;
28  uint256 rentingTime = (now - start);
29  return price * rentingTime;}
30  function returnItem()
31  inState(State.AwaitPayment)
32  onlyBuyer
33  require(msg.value >= rentingFee)
34  payable{
35    itemReturned();
36    state = State.Inactive;
37    uint change = rentingFee - msg.value;
38    if (!buyer.send(deposit + change)||!
    seller.send(this.balance)) throw;}
40  function again
41  function reclaimItem()
42  onlySeller
43  payable
44  require(state == State.Locked || state ==
    State.AwaitPayment){
45    rentingFee = calculateRentingFee();
46    paymentRequested();
47    state = State.AwaitPayment;}}
```

6.2 Ethereum Client and SC Integration

A full Ethereum client on a remote machine is used to integrate the Android client with the BC. There are

several Ethereum client implementations available. Go-Ethereum [27], the Google's Go implementation called Go-Ethereum (Geth) is used in this Dapp.

In this Dapp, Web3j Java library [43] is used to deploy SC on the BC and to interact with SCs from the Android client. Web3j implements the JSON-RPC interface of the Ethereum client and provides high-level access to a transaction and its content. This library supports both local and remote transaction signing and provides tools to generate Java classes that wrap the interface of Solidity contracts and therefore significantly simplifies the development of Ethereum Dapps in Java. Java library contains wrapper classes to deploy SC on the BC and to execute transactions on them. It also contains a data access layer used to manage user accounts and stored contracts on the local Android file system and provides this functionality through different service classes that are briefly described in the next subsections.

6.2.1 ContractService

When deploying or loading a contract object, a lot of parameters have to be provided to the factory method. Since the general parameters, like the Web3j client or the gas price will rarely change, the *ContractService* provides methods for deploying and loading purchase and rental contracts by only specifying the relevant arguments like address of a contract and its constructor arguments. Further, the *ContractService* provides an interface for saving and removing contracts from the local file system.

6.2.2 AccountService

The *AccountService* is responsible for managing accounts and their associated user profiles and provides methods for loading, creating and unlocking accounts. It uses the Web3j client to retrieve general account information from the BC, like the balance of an account. There are two different implementations of the AccountService:

- The *ParityAccountService* uses the *Parity* client of Web3j to unlock accounts on the Ethereum client. Transactions are then signed by the Ethereum client. This implementation only used for testing purposes since it involves sending the password for the wallet file in plain text over the network.
- The *WalletAccountService* uses the *WalletUtil* class of Web3j to create and unlock local wallet files on the Android device. Credentials (private/public key pairs) are then used to sign transactions locally using the *RawTransactionManager* of web3j. Signed transactions are then sent to the Ethereum client. This implementation is secure since the password for wallet file is not sent over the network but it is also much slower because decrypting the wallet file on the Android device can take much time. It takes approximately 30 seconds with weak encryption and 2 minutes with strong encryption to unlock the wallet file on a Samsung Galaxy A5 device.

6.2.3 ConnectionService

The *ConnectionService* is responsible for periodically checking the connection to the Ethereum client and for notifying subscribers when the state of a connection changes. This service provides methods to start and stop polling and to query a connection's state to the Ethereum client.

6.2.4 EthConvertService

The *EthConvertService* provides methods to gather realtime information about *Ether* exchange rates for different currencies and for converting currencies from/to *Ether*. Its implementation uses the RESTful web API of *cryptocompare.com* [14] to retrieve the exchange rates in a JSON format.

6.3 Android client

The Android client provides the user interface for the Dapp. It uses the contract wrapper classes to interact with contracts on the BC and it uses service instances to manage contracts and accounts. The Promise-API is used to dynamically update the UI when transactions completed. Implementation in client side is composed of multiple Android Activity classes that share a common base class. Most UI logic is not implemented in the Activities themselves, but in Fragment classes that can be reused in different contexts.

6.3.1 The ActivityBase class

The ActivityBase class is the base class for all major Activities of the Android client and contains code that is used in every Activity:

- It initializes the Android Toolbar and contains the Toolbar interaction logic
- It handles permission request results (e.g., for accessing the device camera or the external storage)
- It handles completed transactions and other global messages
- It implements the ApplicationContextProvider interface to provide the ApplicationContext (section 6.3.2) to the UI components.

6.3.2 ApplicationContext

The implemented Dapp uses a custom app instance to manage functionality that must be accessible by the UI components including functionality to access the service layer, app settings, Android permissions, and Wifi-P2P interface of a device. It provides these objects through the *ApplicationContext* interface that can be accessed through the *ApplicationContextProvider* interface that is implemented on the *ActivityBase* class.

6.3.3 Activities

Implemented Android client is composed of 6 Main Activities:

- **Account Activity** provides a User Interface (UI) for accessing, creating and managing Ethereum accounts. It displays a list of either remote or local accounts which users can unlock.
- **Overview Activity** displays a list of purchase- or rental contracts that belong to an unlocked account. By selecting a contract item, the details for the contract are displayed in the Detail Activity. The Overview Activity also provides the user interface for importing contracts either by scanning QR-codes or by using the WiFi-Direct channel.
- **Create Activity** provides a UI to specify and create new rental- or purchase contracts. Users can specify price or renting fee of an item together with a textual description and a set of images. Before deploying the contract, users can also decide whether the exchange of personal information between the parties is required and whether they want to deploy all details of a contract on BC or whether they want to use the light deployment mode.
- **Detail Activity** displays the details of a deployed contract to the user. Depending on the state of the contract and the role of the user in the contract, it also displays controls to execute transactions. The Detail Activity also provides a UI to scan the profile of the other contract party and displays the profile details in a separate tab.
- **Profile Activity** provides a UI to view and edit account details such as name, address, and contact details of a user as well as a profile image. The textual details of the profile are displayed as a QR-image that can be scanned by another party to import the profile.
- In **Setting Activity** user can configure global settings, like endpoint of the Ethereum client, whether she wants to use local or remote accounts and which Ethereum transaction parameters should be used by the Web3j library.

6.4 Device-to-device Communication Implementation

As discussed in chapter 4, WiFi-Direct protocol is the most suitable for the use case of exchanging contract and user information, mainly because of its high throughput required to exchange larger media files. Android framework provides the *WiFi-P2P* interface that allows devices to connect directly to each other via WiFi-Direct without an intermediate access point [30].

6.4.1 Connection Management

WifiP2PManager class of Android WiFi-Direct framework is used to discover other peers in the surrounding and to connect to them. The *WifiP2PManager* also allows a client to register callbacks that are invoked when a method succeeds or fails. Further, it can receive intents that notify the client when specific events occur on the framework, e.g., when a new device is detected or when a connection is established or lost [30].

As presented in listing 1.4, the *WifiConnectionManager* class implements the *P2PConnectionManager* interface and uses an instance of the *WifiP2PManager* to discover new peers and to connect and disconnect to them. It further accepts a *P2PConnectionListener* callback that is invoked when a connection is established or lost, when available devices in the surrounding change or when an error occurs during a connection attempt. The purpose of these interfaces is to decouple the P2P connection interface from the underlying protocol implementation.

Listing 1.4: The P2P connection interfaces

```
public interface P2PConnectionManager{
void startListening(P2PConnectionListener CL);
void stopListening();
void connect(String deviceName);
void disconnect();
}

public interface P2PConnectionListener {
void onConnectionLost();
void onPeersChanged(List<String> deviceList);
void onConnectionEstablished(ConnectionInfo CI);
void onConnectionError(String message);
}
```

6.4.2 P2P Service Layer

To facilitate interaction of Android UI Dialogs with the connection management code and with the data exchange code, two interfaces were implemented to accept a buyer or seller specific callback interface in their *requestConnection* method as shown in listing 1.5.

Listing 1.5: The P2PSellerService and P2PBuyerService interfaces

```
public interface P2PService<T extends
    P2pCallback>
{
void disconnect();
void requestConnection(T callback);
}

public interface P2PSellerService extends
    P2PService<P2pSellerCallback>
{
void connect(String deviceName);
}

public interface P2PBuyerService extends
    P2PService<P2pBuyerCallback> {
}
```

Both *P2Pbuyer* and *P2Pseller* services implement the *P2PConnectionListener* interface and register themselves on the *P2PConnectionManager* interface. Both services also use a buyer or seller specific *Peer* implementation to exchange data with the other device.

The *P2PSellerCallback* and *P2PBuyerCallback* interfaces are implemented by the corresponding Android Dialogs and their callback methods are invoked by the service instance in case of connection specific updates or

errors or directly by the peer instance in case of data related updates (when profile or contract related data is requested or has been received or when communication errors occur).

Listing 1.6: The P2PSellerCallback and P2PBuyerCallback interfaces

```
public interface P2pCallback {
void onP2pInfoMessage(String message);
void onP2pErrorMessage(String message);
void onTransmissionComplete();
}

public interface P2pBuyerCallback extends
    P2pCallback
{
void onContractInfoReceived(ContractInfo
    contractInfo);
void onUserProfileRequested(UserProfileListener
     listener);
}

public interface P2pSellerCallback extends
    P2pCallback
{
void onContractInfoRequested(
    ContractInfoListener listener);
void onUserProfileReceived(UserProfile data);
void onPeersChanged(List<String> deviceNames);
}
```

6.4.3 Data Transfer Layer

The actual data exchange protocol used to exchange contract and profile data is implemented by the buyer- and seller specific *Peer* implementations. These instances are created and started by the service instances after a connection between two peers is established as in listing 1.7.

Listing 1.7: The Peer interface

```
public interface Peer {
void start();
void stop();
}
```

Peer implementations use Java sockets to send and receive serialized JSON objects in case of profile and contract details, or binary data-streams in case of image files. In WiFi-Direct, only one peer takes the role of *Group Owner* (GO). After the connection is established, only IP address of the GO is known to both peers. Since assignment of group ownership is not deterministic, both peers can take the role of a server or a client. After a connection has been established, the GO searches a free local TCP port and waits a moment before opening the server socket such that the other peer can also discover the same free port. After the TCP connection established, peers use the opened sockets for the rest of the data exchange. Detalis of data exchange process is illustrated in figure 6.

The seller peer in first step sends to the buyer peer the serialized contract info object, which also contains optional profile information of the seller. The profile information can contain an optional profile image and the

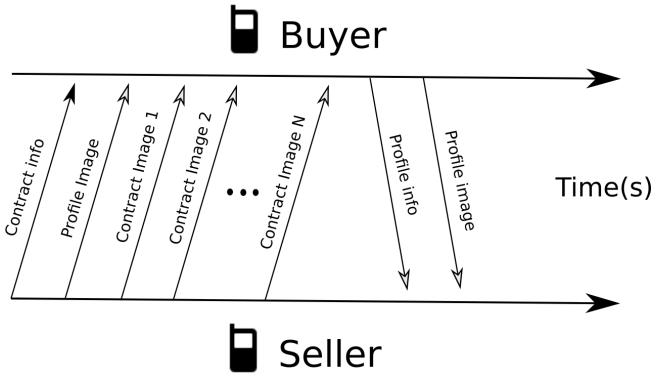

Figure 6: P2P Data Exchange Process.

contract itself can also contain optional images. Since paths of images are contained in the info objects, the buyer peer knows how many image files to expect. If the contract info contains the user profile of the seller, the buyer peer also returns its profile and associated image back to the seller peer.

7 Evaluations

Its crucial to develop SC-based apps with the least possible deploying and transaction costs and for that purpose, first step is to be aware of which parameters of SC codes affect the total cost of deploying contracts and sending transactions.

7.1 Costs

In this section, empirical results of deploying contracts codes are compared to the estimated costs which are calculated with introduced cost estimation functions. Result of these comparisons indicate the high accuracy of provided estimation functions. One general outcome of these evaluations is the difference between light and full contracts which obviously light version has less deployment costs and this explains and proves the incentives of developing light SC.

The actual deployment costs in *gas* for the rental and purchase SC were evaluated using the browser Solidity online compiler [8] and the local Ethereum test client. For the median *gas* price, a value of 21 *gwei (gigawei)* was assumed [25]. For the USD exchange rate, the average value of 210 dollar per *Ether* since May 2017 was taken as a reference value [22].

According to the spreadsheet published in the official Solidity documentation, main costs for deploying a contract are the costs associated with storing the contract code ("CREATEDATA") with cost of 200 *gas* per byte and the costs for storing additional data on the storage of the contract ("STORAGEADD") with 20000 *gas* per 256 bit word. Further, in addition to the 21000 *gas* for a

normal contract transaction, 32000 *gas* have to be paid for a transaction that creates a contract [13].

Deployment costs for a contract depend on the size of the contract code and the amount of bytes that it assigns to the storage in its constructor. Since both the purchase and the rental contract do not perform any calculation intensive work in their constructors, their deployment costs in *gas* can be estimated using the following formula:

$$C_{gas} = (53000 + 200 \times N_{bytes} + 20000 \times N_{words}) \quad (2)$$

Where C_{gas} is the total transaction cost in gas, N_{bytes} is the contract size in bytes and N_{words} is the number of 256 bit words that are initialized in the constructor. To calculate the deployment price in USD, the gas usage has to be multiplied with the the *gas* price P_{gas} and the USD exchange rate for the *Ether* $P_{exchange}$:

$$C_{dollar} = C_{gas} \times P_{gas} \times P_{exchange} 10^{-18} \quad (3)$$

7.1.1 Empirical Results

Table 3 summarizes the measured costs for deploying the rental and purchase contracts in full and light deployment mode using different amounts of additional data (images) to store. The values represent the costs in *gas* and the values in braces represent the costs in USD.

As it is shown in table 3 in all the three tested SC with different number of images as additional data to be stored, costs for light contracts are significantly lower than full contracts as their source code uses less space. Further, they use less storage by storing all content attributes (text and images) in a single 32 byte hash. Therefore, deployment costs are constant and independent of the amount of additional data sent in constructor. Measured values are close to the values estimated by the simplified equations 2 and 3 and are shown in table 4. The Mean Absolute Percent Error (MAPE) is 5.8%.

7.1.2 Transaction Costs

Transaction costs for the purchase and rental contracts are dominated by the fix transaction costs "GTX"of 21000 *gas*, the costs for adding a 256 bit word to the storage "STORAGEADD"of 20000 *gas*, the costs for modifying a word on the storage "STORAGEMOD "of 5000 *gas* and the costs for making a call from the contract that contains *Ether* "GCALLVALUETRANSFER"of 9000 *gas*. Other costs are not significant since no major computation is done in any of the transaction functions. Therefore, to estimate the transaction costs, the formula 4 is used which yields to very good results with the MAPE of only 3.8%.

$$C_{gas} = 21000 + 20000 * N_{words\_add} + 5000 * N_{words\_mod} + 9000 * N_{tx} \quad (4)$$

Table 3: Measured deployment costs for different deployment configurations

	Empty Contract	With One Image (32bytes)	With Two Images (64bytes)	With Three Images (96bytes)
Purchase Contract	691023 (3.05)	697617 (3.08)	719104 (3.17)	740591 (3.27)
Rental Contract	760550 (3.35)	797144 (3.52)	818631 (3.61)	840118 (3.70)
Purchase Contract-light	389702 (1.72)	389702 (1.72)	389702 (1.72)	389702 (1.72)
Rental Contract-light	486946 (2.15)	486946 (2.15)	486946 (2.15)	486946 (2.15)

Table 4: Estimated deployment costs for different deployment configurations

	Empty Contract	With One Image (32bytes)	With Two Images (64bytes)	With Three Images (96bytes)
Purchase contract	644800 (2.84)	664800 (2.93)	684800 (3.02)	704800 (3.11)
Rental contract	737000 (3.25)	757000 (3.33)	777000 (3.43)	797000 (3.52)
Purchase contract-light	363600 (1.60)	363600 (1.60)	363600 (1.60)	363600 (1.60)
Rental contract-light	454000 (2.00)	454000 (2.00)	454000 (2.00)	454000 (2.00)

Where C_{gas} is the total cost in gas, N_{words_add} is the number of 256 bit words added to the storage in the transaction, N_{words_mod} is the number of words that are modified in the transaction and N_{tx} is the number of messages with *Ether* that are sent in the transaction function (e.g. for refunding *Ether* to the buyer or seller). The total costs in USD can be estimated by equation 3.

Table 5 compares the measured transaction costs for all transactions of the rent and purchase contracts to the estimated costs with equation 4. It was found that a variable is only initialized on the storage when a value different from 0 is assigned to it. For example, the *state* variable of the contract is only added in the *abort* or *confirmPurchase* function, when the value changes from the initial state. This has to be taken into account when using the formula.

7.2 Application Scalability

There are two factors that limit the number of contracts that this Dapp can handle concurrently:

Table 5: Real and estimated transaction costs

	abort	confirm purchase	confirm received	rent item	reclaim item	return item
Measured costs	49584 (0.22)	62890 (0.28)	41657 (0.18)	82721 (0.36)	41913 (0.18)	48146 (0.21)
Estimated costs	50000 (0.22)	61000 (0.27)	44000 (0.19)	81000 (0.36)	44000 (0.19)	45000 (0.20)

- **Internal storage limit**: Since every contract is saved on the device, the available free space on the internal storage limits the number of contracts that can be stored.
- **Synchronization Overhead**: Since every loaded contract is synchronized with the BC through the Ethereum client on the server, the polling requests needed to keep track of the contract states could lead eventually to a very high CPU and network load on the Android device.

7.2.1 Internal Storage Limit

The internal storage limit available for the application depends on the total size of the internal storage and the internal storage space that is occupied by other apps. The size of one contract depends on the number of characters and on the number of images used for the description of a contract item. A minimum JSON serialized contract that stores only 1 character as title, one character as description plus the metadata needed to properly load the contract from the BC (address, contract type) has a size of 179 bytes. Therefore the upper limit for the number of contracts that can be stored on the internal storage can be expressed with the formula:

$$N_{contracts} \leq (S_{internal} - S_{occupied}) \tag{5}$$

Where $N_{contracts}$ is the number of contracts, $S_{internal}$ is the total size of the internal storage in bytes and $S_{occupied}$ is the internal storage in bytes that is occupied by other apps.

7.2.2 Synchronization Overhead

The synchronization overhead is not a limiting factor in reality. The only Activity that loads and displays more than one contract, is the *OverviewActivity* (section 6.4.2). It uses an Android *RecyclerView* that can reuses already instantiated views again when the user scrolls down in the list. Every time a view is bound the a contract at another position, the view is unregistered from the old contract and registered on the new contract. A contract is registered for all events that can happen on the SC on the BC and every event subscription involves 1 HTTP request to the Ethereum client to check if the event has been emitted (see section 6.3.2). Debugging the Dapp revealed that only four different views are used to display the contracts and therefore only the states for four contracts have to be tracked on the Ethereum client concurrently. In an empirical test that loaded 1000 contracts into the *RecyclerView*, no performance issues were detected. When scrolling through the list, contracts are loaded continuously.

7.3 Privacy

This section discusses how privacy issues that arise from storing and exchanging personal user data are addressed in the Dapp. It further discusses possible privacy issues that can rise when storing contract details on the BC and how they can be avoided.

7.3.1 Privacy of Personal User Data

All profile information including profile images are stored on internal storage of Android app to prevent other apps on the device access this data. However, it does not provide protection against an attacker that has physical access to a non-encrypted file system or against an attacker that has root access on the operating system [2]. When user data is exchanged over WiFi-Direct it is always encrypted using WiFi Protected Setup (WPS). As discussed in section 5.3.4, WPS uses WPA2 to encrypt and authenticate data and offers good protection against eavesdropping and brute-force attacks but it is vulnerable against MITM attacks during the short time frame in which the encryption keys are exchanged.

7.3.2 Privacy of Contracts

Storing contract details in plain text on a SC can cause privacy issues because the addresses of the seller and the buyer are also stored on the contract and are therefore publicly accessible. A party that knows a person with a particular address (e.g., when it exchanged contract or user data with that person in the past) can look up details of other contracts that were signed by that person. To prevent that, the light deployment option can be used. When using light deployment, only the hash of the contract details that do not have to be stored on the BC is stored in the SC.

7.4 Security

This section discusses the most important security aspects of Ethereum accounts and transactions.In the proposed app, accounts can either be managed locally on the Android device or remotely on the server running the Ethereum client. When using the *WalletAccountService* the local wallet file is decrypted using the password provided by the user and the credentials are used by the *RawTransactionManager* to sign every transaction with the secret key (SK) belonging to the account. *WalletAccountService* provides protection against eavesdropping since the wallet password is never sent to the Ethereum client. It also provides protection against MITM attacks since a transaction cannot be altered any more after it has been signed with the SK. Password for a wallet file

is never persisted on the file system and the SK of an unlocked wallet file is only stored in volatile memory.

There are only two scenarios left in which an attacker could obtain or use the SK of an account:

- The attacker has root access on the operating system and can intercept the password from the user by using a key logger.
- The attacker has physical access to the phone and the account is still unlocked. In this case the attacker could use the account to sign rent or purchase contracts and transmit money to her own account.

8 Conclusions

Conventionally, setting up purchase or rental contracts between two parties requires providing complete and accurate identity information of the both parties besides the details of exchanged subjects. Hitherto proposals and apps have followed a centralized approach to ease the required steps to be taken. However, these apps inherit the disadvantageous of centralized architectures. Most important challenges in designing such apps are the P2P communications between parties for transferring contract information, providing identity information while ensuring user privacy, legal binding, creating trust in the system, and data security. To avoid all the paper work required for setting up legally valid purchase and rental contracts between two parties and to tackle the problems in centralized apps, this paper proposed a SC-based Dapp.

The presented work proposed design and implementation of an Android-based, SC-based, and decentralized app with the goal of satisfying the functional and legal requirements of an automated purchase and rental contracts setting Dapp in adoption to Ethereum BC. This system is designed with a P2P architecture which, enables high data rates in D2D communications by using WiFi-Direct. In addition to the design of a full contract, proposed Dapp introduced the light-weight contracts which enable the opportunity of transferring identity information between users without the need to store them all in the public BC and therefore reduce the costs.

Trust to this system is provided by designing the deposit-based SC to guarantee seller and buyers loyalty to contract and money transfer. High privacy provided by design as this Dapp is completely independent of any TTP and seller can specify the contract details including price, deposit, textual description, and images to deploy a contract. Contract details can be exchanged either by scanning a QR-code of a contract or by using WiFi-Direct. In the latter case, parties can flexibly decide which personal information they want to share. Contracts are signed on the users local device and protected by storing only on internal storage and being encrypted while sent over the network. Proposed Dapp is highly scalable and is capable of handling large number of contracts in parallel. Also, it detects network errors and prevents lose of money by storing contracts pre-emptively when the network fails during contract creation transactions.

Evaluations indicate that cost estimation equations proposed in this paper are accurate and estimate the deploying and transactions of SC costs with high precision. These equations lead us to produce the SC codes with the least possible costs.

Literature

[1] Airbnb. URL:https://www.airbnb.com, Last visited September 28, 2017

[2] Android, Security Tipps; URL:https://developer.android.com/training/articles/securitytips.html, Last visit July 31, 2017.

[3] P. Arana: *Benefits and Vulnerabilities of Wi-Fi Protected Access 2 (WPA2);* URL:http://cs.gmu.edu/~yhwang1/INFS612/Sample_Projects/Fall_06_GPN_6_Final_Report.pdf, Last visit July 30, 2017.

[4] *Bitcoin, Proof of Work*; URL:https://en.bitcoin.it/wiki/Proof_of_work, Last visit May 9, 2017.

[5] *Bluetooth Radio Interface, Modulation and Channels*; URL:http://www.radio-electronics.com/info/wireless/bluetooth/radio-interface-modulation.php, Last visit July 21, 2017.

[6] A.Bogner, M.Chanson, A.Meeuw: *A Decentralised Sharing App running a Smart Contract on the Ethereum Blockchain;* 6th International Conference on the Internet of Things(IoT16), Stuttgart, Germany, November 2016.

[7] *Brooklyn Microgrid;* URL:http://microgridmedia.com/brooklyn-startup-broadens-solar-power\-access-with-p2p-energy-exchange/, Last visit May 9, 2017.

[8] *Browser Solidity Online Compiler*; URL:https://ethereum.github.io/browser-solidity/, Last visit September 19, 2017.

[9] E.Bucher:*Zustandekommen des Vertrages*; URL:http://www.eugenbucher.ch/pdf_files/Bucher_ORAT_10.pdf, Last visit May 9, 2017.

[10] E.Bucher: *Kaufvertrag im Allgemeinen.* URL:http://www.eugenbucher.ch/pdf_files/Bucher_ORBT_03.pdf, Last visit May 9, 2017.

[11] V.Buterin: *A next-generation smart contract and decentralized application platform.* White paper, 2014.

[12] D.Camps-Murr, A.Garcia-Saavedra, P.Serrano: *Device to device communications with WiFi Direct: overview and experimentation.* URL:http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=81D8D0CEA0130A332E2EB14EC8563A1E?doi=10.1.1.725.7590&rep=rep1&type=pdf, Last visit July 20, 2017.

[13] CryptoCompare.com: URL:https://www.cryptocompare.com/, Last visit May 9, 2017.

[14] CryptoCompare.com. URL:https://www.cryptocompare.com/, Last visited May 9, 2017

[15] *Cryptocurrency Market Capitalization;* URL:https://coinmarketcap.com/ Last visit May 9, 2017.

[16] K. Delmolino, M. Arnett, A.E. Kosba, A. Miller, E. Shi: *Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab.* IACR Cryptology ePrint Archive, 2015 (2015), 460, URL:http://fc16.ifca.ai/bitcoin/papers/DAKMS16.pdf, Last visited August 12, 2017

[17] *Digix White paper;* URL:https://dgx.io/whitepaper.pdf, Last visit May 9, 2017.

[18] Ebay. URL:http://www.ebay.com, Last visited September 28, 2017

[19] *Ebay, general terms of agreement;* URL:http://pages.ebay.ch/help/policies/user-agreement.html, Last visit May 9, 2017.

[20] Ethereum, Documentation. URL:http://www.ethdocs.org/en/latest/, Last visited May 9, 2017.

[21] Ethereum.io: *Safe Remote Purchase;* http://solidity.readthedocs.io/en/latest/solidity-by-example.html#safe-remote-purchase, Last visit September 19, 2017.

[22] *Ethereum, Average Exchange Rate Since May 1, 2017;* URL:https://poloniex.com/public?command=returnChartData&currencyPair=USDT_ETH&start=1493596800&end=1501286400&period=86400", Last visited July 28, 2017.

[23] Ethereum, Solidity. URL:https://solidity.readthedocs.io/en/develop/, Last visited May 9, 2017

[24] *Ethereum, Solidity, Global variables and functions;* URL:https://solidity.readthedocs.io/en/latest/units-and-global-variables.html, Last visit May 9, 2017.

[25] Ethgasstation: URL:https://ethgasstation.info, Last visited July, 16, 2017.

[26] *Federal law about the electronic signature (ZertES);* URL:https://www.admin.ch/opc/de/classified-compilation/20011277/index.html, Last visit May 9, 2017.

[27] Go-Ethereum: URL:https://github.com/ethereum/go-ethereum, Last visited May 9, 2017.

[28] K.Haataja, K.Hyppönen, S.Pasanen, P.Toivanen: *Bluetooth Security Attacks, Överview of Bluetooth Security";* SpringerBriefs in Computer Science, 2013. URL:http://www.springer.com/cda/content/document/cda_downloaddocument/9783642406454-c2.pdf?SGWID=0-0-45-1434420-p175453762, Last visit August 12, 2017.

[29] F.Hawlitschek, T.Teubner, G.Henner: *Understanding the Sharing Economy - Drivers and Impediments for Participation in Peer-to-Peer Rental;* 49th Hawaii International Conference on System Sciences(HICSS), Koloa, HI, USA, Januar 2016, ISBN: 978-0-7695-5670-3.

[30] *IEEE 802.11 Wi-Fi Standards*; URL:http://www.radio-electronics.com/info/wireless/wi-fi/ieee-802-11-standards-tutorial.php, Last visit July 21, 2017.

[31] JDeffered; URL:https://github.com/jdeferred/jdeferred, Last visited May 9, 2017

[32] JQuery; URL:https://github.com/jquery/jquery, Last visited May 9, 2017

[33] S.Nakamoto: *Bitcoin: A peer-to-peer electronic cash system [2008];* URL:https://bitcoin.org/bitcoin.pdf, Last visited May 9, 2017.

[34] D. Primavera: *The interplay between decentralization and privacy: The case of blockchain technologies;* http://peerproduction.net/issues/issue-9-alternative-internets/peer-reviewed-papers/, Last visit September 28, 2017.

[35] *QrGen*; URL:https://github.com/kenglxn/QRGen, Last visit May 9, 2017.

[36] Relayrides. URL:https://www.relayrides.com, Last visited September 28, 2017

[37] *Shocard Whitepaper;* URL:https://shocard.com/wp-content/uploads/2016/11/travel-identity-of-the-future.pdf Last visit May 9, 2017.

[38] *SuisseId Multi Signing Platform;* URL:https://www.multisigning.ch/, Last visit August 4, 2017.

[39] Swiss Federal Council: *Federal Law on the Supplement to the Swiss Civil Code;* URL:https://www.admin.ch/opc/de/classified-compilation/19110009/index.html#a1, Last visited Sep 28, 2017

[40] N.Szabo: The idea of smart contracts [1997]; URL:http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_idea.html, Last visited May 9, 2017.

[41] *Transactive Grid;* URL:https://www.slideshare.net/JohnLilic/transactive-grid, Last visit May 9, 2017.

[42] S.Viehböck: Brute forcing Wi-Fi Protected Setup; URL:https://sviehb.files.wordpress.com/2011/12/viehboeck_wps.pdf, Last visited July 11, 2017.

[43] *Web3j library;* URL:https://github.com/web3j/web3j, Last visit September 22, 2017.