Blockchain Report

HouseLocker Team
William R. Duro, Anna Giacomello, Davide Pasetto, Federico Valbusa

July 10, 2023

Abstract

In this report, we will describe how we tried to solve a practical problem through the use of blockchain. In particular, we are looking at someone who is trying to rent a room away from where they live. In the report, we will refer to them as *student* since this happens quite often with off-site students.
Suppose the student has seen a nice house or room online and wants to ensure they will rent exactly that room. Nowadays it happens that those who want to rent a room must send some money in advance to the householder to "block" the room, showing their real interest. This process relies completely on the fact that the student must trust the landlord, who potentially could steal the amount of money sent to them.
With our idea, we try to automatize this process avoiding the need for trust with the transparency of the blockchain.

Contents

1 General Business Idea

Our innovative business offers a unique solution to a common problem faced by students seeking to rent a room: the lack of trust in the landlord. We provide a trustless approach using cutting-edge blockchain technology and smart contracts. Our platform allows homeowners to list their rooms for rent and agree to use our secure blockchain technology. Students can then search for their ideal room on our app or website and initiate a smart contract with the landlord. The landlord accepts the contract and the student's payment is held securely in the smart contract. The landlord also puts down a deposit to guarantee their good faith. Once both parties agree that the contract has been fulfilled correctly, the landlord receives their deposit and the student's fee. Our solution provides peace of mind for both landlords and students, ensuring a secure and transparent rental process.

1.1 Problem description

The phenomenon of out-of-town students is constantly growing and more and more people are leaving their region of origin (or even their country) to study at universities that are more attractive for their studies. One of the most stressful things to do when embarking on this path is to look for a house or, more often, a room in the city where they will live. The student room market is a flourishing market that is steadily growing and shows no signs of decreasing. In such a context, it is obvious that ill-intentioned people may try to take advantage of the physiological naivety of young adults. It is not uncommon to have direct or indirect experiences of people who have been scammed by some malicious homeowner. One of the most recurring situations is that of a person who, while they are still in their home-town, comes across an attractive room for which he would be very interested in signing a contract, the owner of the property then asks him for a sum of money that serves to "lock" the room, i.e. to guarantee that the owner will not rent it out to anyone else before the student makes a decision. With the condition that in case of withdrawal, the owner will keep the sum as compensation for the time lost. The problem with this system is that it is based solely on trust between the people and there is no legal way to conclude such a contract remotely or payment services that provide certainty in not being cheated. In this context, the need for a solution that gives more guarantees to students is desirable and this is what we aim for.

1.2 Solution

Our blockchain company aims to solve this problem with smart contracts. Smart contracts are self-executing and self-verifying digital contracts based on blockchain technology that allows contract execution to be automated without intermediaries.

In this way, smart contracts can guarantee the security and reliability of online transactions. Thanks to smart contracts students will be able to conclude digital

contracts with owners without having to blindly trust the owner himself.

We provide a service that enables a student to lock a house or a room in an effortless and transparent way, without the need of trusting anyone.

Houses and rooms information is visible in the web app we developed. After a student has chosen the best room available according to their personal need, a smart contract instance is initiated. It will interact with the student and the landlord accounts on the blockchain.

The contract is designed to deter malevolent behaviors both from students and landlords. In particular, the transparency of blockchain structure marks every type of malicious action. Moreover, the cryptographic properties of blockchain allow the verification of any address, and so the correct and valid interaction between parties, without disclosing any private information. Our service is more than just a rental platform - we are selling a dream of hassle-free, secure renting for students and homeowners.

2 Assumptions

As stated before, we are going to provide a trustless, secure environment for students and landlords to interact in the most transparent way possible. Unfortunately, some pieces of information need to be verified or provided by external parties to allow a secure and correct functioning of our service.

2.1 Data about rooms and houses

Since the students and the landlords make agreements upon physical assets, i.e. the rooms to rent, which are outside our field of action, we need to rely on external partnerships to assess the properties of the houses or rooms they are going to put up for rent. We decided to opt for a partnership with an institutional party that will be in charge of visiting the landlord's property and providing information about it, for instance, the owner's name, the square footage of the rooms, the appliances provided by the owner, the energetic class of the house and so on. The non-sensible components of this data, along with the landlord's public key, will be stored in a public database and therefore will be accessible to anyone.
During the registration phase, the application will be in charge of performing the following steps:

- Running an account verification (5.3.1) to make sure that the user is the owner of the address they are providing
- Retrieving the rooms' information from the database using the public key generated in the account verification phase
- Transposing the rooms' data in the application so that they will be available to the students browsing the application

Although we understand that we are putting faith in an institutional third party in this step, we also believe that the existence of a public database will still partially ensure transparency about properties' data.

2.2 Verification of students' status

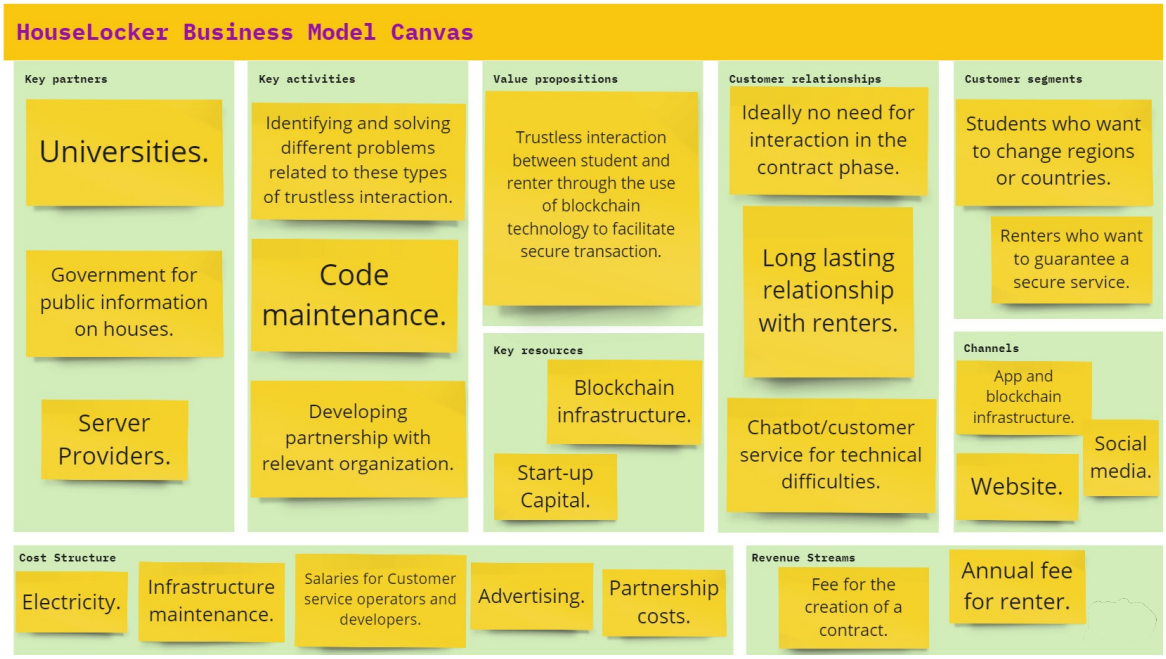
When a user registers as a student, the application needs to make sure that the user is, indeed, a student. This necessity arises because housing contracts for students are different from long-term ones for workers and they benefit from a more agile structure. The verification of students' status is performed with the use of zero-knowledge proof. In particular, we suppose the existence of a public database with students' data encrypted. This is a realistic assumption since such databases are necessary for password verification in authentication procedures.

The zero-knowledge proof works as described in section 5.3. In particular, the public database contains the password of students' accounts encrypted with discrete logarithms on elliptic curves. We treat this data in the same way as the blockchain accounts' public keys. If a student can produce valid proof for one of the encrypted values in the public database, they must possess the private value which yields the public one when encrypted, so they must be a student.

2.3 Blockchain choice

Since the application is responsible of running checks before allowing some of the contract functionalities (e.g. the registration of the rooms) we need to guarantee the app is the only point of entry to the contract.
That is the reason why we decided to opt for a private blockchain.

3 Business Model Canvas



- Key partners:
  - Universities will provide some encrypted information about students, used to verify their status.
  - State services will provide information on houses and rooms.
  - Server Providers will allow us to use blockchain to guarantee our service.
- Key activities:
  - We will improve the code to fix possible bugs, integrate new functionalities, and stay up to date.
  - We are ready to collaborate with some already existing rental companies, adapting their business to our model.
- Value propositions:
  - We will provide trustless and transparent interactions between students and landlords through blockchain.
- Key resources:
  - We need staff to maintain servers and blockchain infrastructure.
  - We also need initial investments to build the infrastructure and start the business.

- Customer relationships:
  - Our service highly encourages correct behavior from both parties. There are no good reasons for both the landlord and the student to act maliciously. In case of disputes, support to go to the competent authorities.
  - It is supposed that students will use our service a limited number of times. landlords instead can repeatedly use it to guarantee the house is locked in a trusted and transparent way.
  - We also have to take care of possible technical difficulties encountered by the clients with the help of a chatbot.
- Customer segments:
  - University students, especially those who want to change region or country.
  - Landlords, who will guarantee a secure service.
- Channels:
  - Website for browsing houses.
  - Social networks for advertising.
  - DApp and blockchain infrastructure for contract operations.
- Cost structure:
  - Electricity for infrastructure use.
  - Employees: developers and customer service.
  - Advertising to find clients.
  - Partnership with other companies to improve our range of clients.
- Revenue Streams:
  - After the first successful usage of our service landlords will pay an annual subscription.
  - Students will pay the cost established by the landlord to lock the room and will pay us an additional 10% of that value for the initialization of the contract instance.

4 Contract functioning

In this section, we are going to discuss the functioning of the contract at a high level.

The user experience begins with what we can call a setup phase: landlords and students can register, the contract runs some checks about the user to make sure they are not providing false information about their identity, and landlords can upload information about their rooms.
Then, when a student finds a room that is suitable for their needs, they initialize a contract instance. Here begins the "Initial" phase of the contract.

4.1 Initial phase

From the moment the contract instance is initialized, the parties involved have a safe window (which we set to 48 hours) in which they can withdraw from the contract without any penalty; this means that, in case they decide to withdraw, they get their money back (except for the additional 10% on the deposit potentially paid by the student).
The moment both parties pay their deposit, the contract is considered to be "Stable".

If one or both parties don't pay their deposit by the deadline, i.e. 48 hours after the contract initialization, the contract instance is destroyed and the money potentially paid by one of the parties is given back to them (except for the additional 10% of the deposit potentially paid by the student).

4.2 Stable phase

In this phase not much happens: the contract just exists, guaranteeing a neutral and secure place to store both parties' deposits.

If one party wishes to withdraw from the contract, the contract instance is deleted and the deposits from both parties (except for the additional 10% on the student's deposit) are given to the other party as a refund for the inconvenience.

The moment the landlord and the student meet in person and they sign the legal lease, they communicate to the contract that they consider it to be concluded successfully. The contract instance is deleted (but the room keeps figuring as occupied to avoid double booking) and all the money deposited in the contract (except for the additional 10% on the student's deposit) is given to the landlord. That is because the student's deposit will also serve as the legal deposit for the lease.

Since our main goal is to provide guarantees for both parties involved in the contract, we designed a particular protocol in case one party suspects the other of malevolent behavior.

If that's the case, the wronged party can hibernate the contract, so that it becomes "frozen" until an external authority determines who behaved unfairly.

4.3 Hibernated phase

The moment a contract instance gets hibernated, neither party can withdraw from it. It is also worth noting that this status can only be reached if the contract was previously in the "Stable" phase.

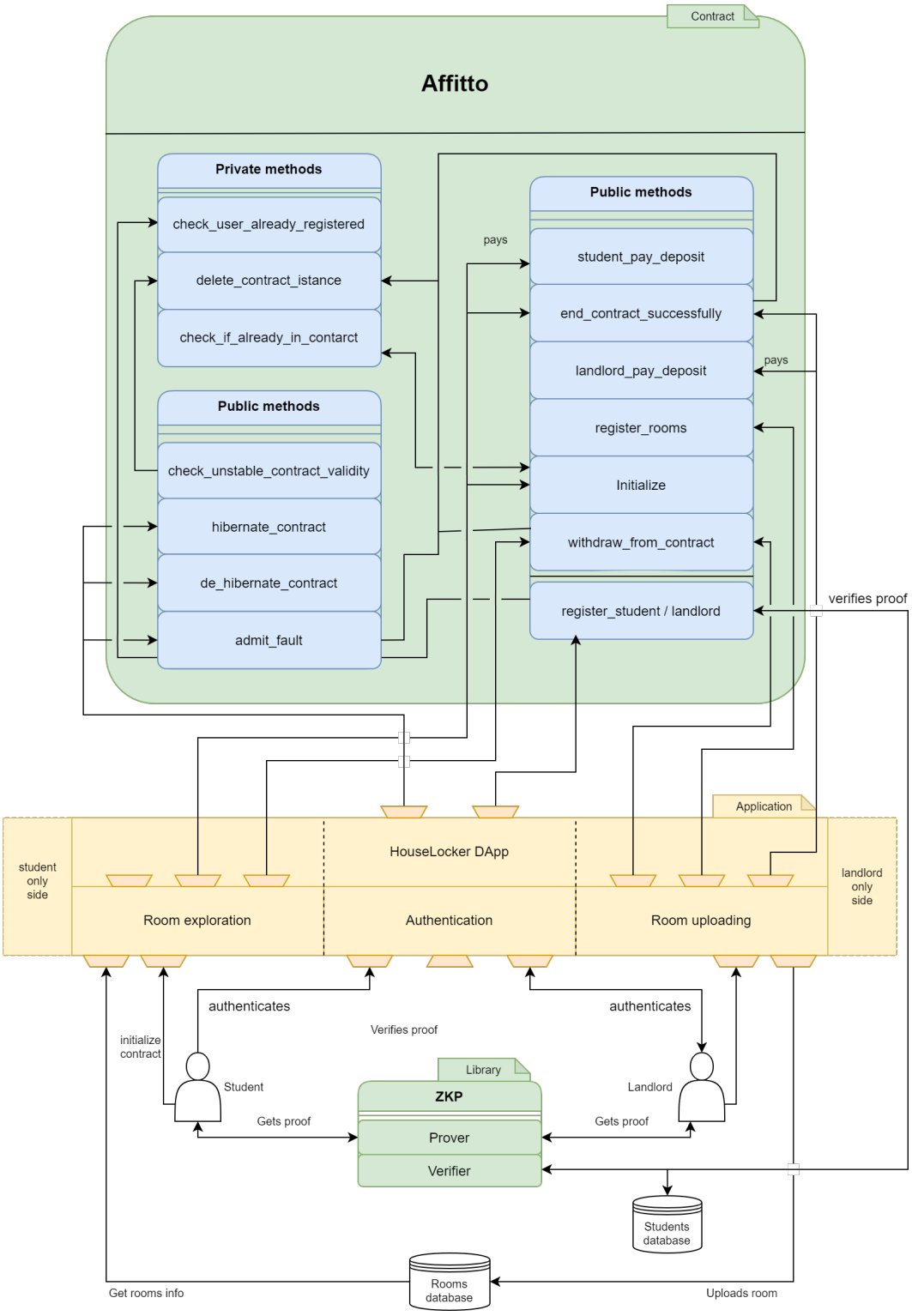
If the parties manage to reach a peaceful resolution, then the party that hibernated the contract can restore its previous status of stable contract and it can continue to serve its purpose.

If one party eventually admits fault, then the contract instance is deleted and the deposits from both parties (except for the additional 10% on the student's deposit) are given to the other party as a refund for the inconvenience.

5 Implementation Choices

In this section, we give a more detailed explanation of the main function and contract. In particular, the first part describes the Main Contract that we called, to be parochial, "Affitto". This involves all the operations regarding registering students and landlords and creating a contract between the two parties. A description of how the various functions work and how we decide to save the important data regarding the members will be given. In the second part, we describe our adaptation of a zero-knowledge algorithm together with the description of the code and a description of some concrete use cases alongside different possible implementations.

5.1 UML diagram



5.2 Affitto

The description is divided into two parts. First, we describe all the variables used and what they represent, then we describe all the functions and how they work.

5.2.1 Variables

The contract defines several structs: *contract_instance*, *room*, and *user*.

The *contract_instance* struct represents an instance of a rental contract between a student and a landlord for a specific room. It has several fields to keep track of the contracts state, such as whether the student and landlord have paid their deposits, the amount paid by each party, whether each party considers the contract as concluded successfully, and the phase the contract is currently in.

The *room* struct represents a room that can be rented. It has fields for the room ID, owner, occupancy status, and deposit amount.

The *user* struct represents a user of the contract. It has fields for the user's role (either student or landlord), an array of contract IDs that represents the contract instances the user is currently a part of (a student can only be in one contract at most), the number of rooms associated with the user (a student can't own any), and a boolean flag to indicate whether the user has been initialized.

An enum named *Role* with two values: *Landlord* and *Student*. This is used to represent the role of a user in the contract.

Another enum called *ContractPhase* describes the status a contract can have: Initial. Stable or Hibernated.

Several mapping variables map keys to values: *contract_record* maps contract IDs to *contract_instance* structs, *user_info* maps user addresses to user structs, *landlords_rooms* maps landlord addresses to arrays of room IDs owned by the landlord, and *rooms_record* maps room IDs to room structs. *hibernated_contracts* maps the contract id of the hibernated contracts to the address of the user that hibernated them.

Various arrays hold the keys for these mappings: *contract_record*_keys, *user_info_keys*, *landlords_rooms_keys*, and *rooms_record_keys*.

A uint variable named *conversion_rate* holds the conversion rate between Ether and some other currency (used for calculating deposit amounts).

A uint variable named *num_contracts* keeps track of the number of contracts created by the contract.

A uint variable named *landlord_deposit* sets the amount of money a landlord has to pay for the deposit.

A uint variable named *time_limit_to_make_contract_stable* sets the maximum amount of time a contract instance can stay in the "Initial" phase. Past that limit, if the contract instance has not become "Stable" it gets deleted.

Finally, the contract has a **constructor** that initializes the *num_contracts* variable to 1; we decided to start from 1 so we could reserve 0 as a default value when needed It also initializes the landlord deposit to 100 and the maximum amount of time to make

a contract stable to 48 hours (converted to seconds).

5.2.2 Functions

Now we will describe the several functions of the contract.

The **register_landlord** function allows a user to register as a landlord. It takes as arguments six uint arguments used for verifying that the user is indeed the owner of their address using zero-knowledge proofs (the details of this verification are described later in the report). The function checks that the user is indeed the owner of their address using zero-knowledge proofs and that they have not already registered. If these checks are successful, it creates a new user struct for the user with their role set appropriately and adds it to the *user_info* mapping.

The **register_student** function works in the same way as **register_landlord**, but this time it takes six additional uint arguments to run an additional zkp to make sure that the student is indeed a student.

The **register_rooms** function allows a landlord to register rooms that they own. It takes two arguments: an array of room IDs and an array of deposit amounts (one for each room). The function checks that the sender is indeed a landlord and that they have provided consistent information (i.e., that the lengths of the two arrays match). If these checks pass, it creates new room struct for each room with their ID, owner (set to the sender), occupancy status (set to false), and deposit amount set appropriately and adds them to both the *rooms_record* mapping and the appropriate entry in the *landlords_rooms* mapping.

The **initialize** function allows a student to initialize a rental contract with a specific landlord for a specific room. It takes two arguments: an address representing the landlord and a uint representing the ID of the room. The function checks that both parties are registered users, that neither party is already in another contract instance, and that the landlord is indeed the owner of the specified room. If these checks pass, it creates a new *contract_instance* struct representing an instance of a rental contract between these two parties for this room with its fields set appropriately (e.g., both parties' deposit status set to false and the contract phase is set to "Initial") and adds it to both the *contract_record* mapping and both parties entries in their respective entries in the *user_info* mapping.

The **student_pay_deposit** function allows a student to pay the deposit for their room. It is payable and takes no arguments. The function checks that the sender is indeed a student and retrieves their contract instance from the *contract_record* mapping. It then calculates the required deposit amount by multiplying the conversion rate by the deposit amount for the room specified in the contract instance. It checks that the value sent with the transaction matches this calculated amount and, if it does, updates the contract instance to indicate that the student has paid their deposit. The function then checks if both parties have paid their deposit: if so, the contract status is set to "Stable".

The **landlord_pay_deposit** function allows a landlord to pay the deposit for their

room. It is payable and takes one argument: a uint representing the ID of their contract instance. The function retrieves this contract instance from the *contract_record* mapping and checks that the sender is indeed the landlord specified in this contract instance. It then calculates the required deposit amount by multiplying the conversion rate by the deposit amount for the room specified in this contract instance. It checks that the value sent with this transaction matches this calculated amount and, if it does, updates this contract instance to indicate that the landlord has paid their deposit. The function then checks if both parties have paid their deposit: if so, the contract status is set to "Stable".

The **withdraw_from_contract** function allows either party to withdraw from their contract. It takes one argument: a uint representing the ID of their contract instance. The function retrieves this contract instance from the *contract_record* mapping and determines which party is withdrawing (the sender) and which party is not (the other party).

Then the function identifies two possible cases. If the contract is still in the initial phase, then any party that has already paid receives the money back (minus the additional 10% on the deposit potentially paid by the student). On the other hand, if the contract is in the stable phase, the money paid by both parties (minus the additional 10% on the deposit paid by the student) is transferred to the party that hasn't withdrawn to make up for the inconvenience. In both cases, at the end, the function calls an internal function named **delete_contract_instance** to delete this contract instance and free up the room.

The **hibernate_contract** function allows one party in a contract to hibernate the contract if they suspect malevolent behavior. It takes *contract_id* as a parameter and checks if the calling party is either the student or the landlord involved in the contract. The function also verifies that the contract is currently in the "Stable" phase. If all the conditions are met, the contract status is changed to "hibernated," and the address of the party who initiated the hibernation is recorded in the *hibernated_contracts* mapping.

The **de_hibernate_contract** function allows the party who initiated the hibernation to unlock and restore the contract to its previous state of being stable. It requires that the calling party is the same as the one who initiated the hibernation for the specified *contract_id*. The function removes the hibernation record from the *hibernated_contracts* mapping and changes the contract status back to "Stable."

The **admit_fault** function allows one party in a hibernated contract to admit fault. It requires that the calling party is either the student or the landlord involved in the contract. Additionally, it checks if the contract phase is "hibernated". If all the conditions are met, the function calculates the total amount to be refunded to the other party, which includes the deposits paid by both parties minus the additional 10% on the student's deposit retained by the contract. The total amount is transferred to the other party, and the contract instance is deleted.

The **check_unstable_contracts_validity** function iterates over the *contract_record* mapping to check the validity of unstable contracts. It takes the *current_timestamp* as a parameter. For each contract in the "Initial" phase, it compares the current timestamp

with the creation time plus the defined time limit to make the contract stable. If the time limit has been exceeded, it means that one or both parties haven't paid their deposit before the deadline and thus the contract has expired. The contract instance is deleted, and any funds paid by either party are returned (minus the additional 10% on the deposit potentially paid by the student). The function returns an array containing the contract IDs of all the deleted contract instances. This function is designed to be called periodically by the application, which will provide the current timestamp, adequately converted to a format readable by the contract.

The **end_contract_successfully** function allows either party to end their contract successfully. It takes one argument: a uint representing the ID of their contract instance. The function retrieves this contract instance from the *contract_record* mapping and determines which party is ending the contract (the function caller) and which party is not (the other person in the contract). It updates the contract instance to indicate that the sender has concluded the contract. If both parties have concluded the contract, it calls an internal function named **delete_contract_instance** to delete this contract instance and transfer any funds paid to the landlord.

The **get_user_info** function allows anyone to retrieve information about a user. It takes one argument: an address representing the user. The function retrieves the user struct for this user from the *user_info* mapping and returns several pieces of information about them: their role (as a boolean, with true representing a landlord and false representing a student), an array of room IDs associated with them, the number of rooms associated with them, and a boolean flag indicating whether they have been initialized.

The **get_room_list** function allows anyone to retrieve a list of room IDs owned by a landlord. It takes one argument: an address representing the landlord. The function retrieves the array of room IDs for this landlord from the *landlords_rooms* mapping and returns it.

The **get_contract_info** function allows anyone to retrieve information about a contract instance. It takes one argument: a uint representing the ID of the contract instance. The function retrieves this contract instance from the *contract_record* mapping and returns several pieces of information about it: its ID, the ID of the room associated with it, whether the student and landlord have paid their deposits, the amount paid by each party, whether each party has concluded the contract and the addresses of both parties.

The **get_deposit_in_wei** function allows anyone to retrieve the deposit amount for a room in wei. It takes one argument: a uint representing the ID of the room. The function retrieves this room from the *rooms_record* mapping, calculates its deposit amount in wei by multiplying its deposit amount by the conversion rate, adds 10% on top of it as the contract profit and returns this calculated amount.

The **get_room_info** function allows anyone to retrieve information about a room. It takes one argument: a uint representing the ID of the room. The function retrieves this room from the *rooms_record* mapping and returns several pieces of information about it: its ID, its owner's address, its occupancy status (as a boolean), and its

deposit amount.

The **check_if_already_paid** function allows anyone to check whether a specific user has already paid their deposit for a specific contract instance. It takes two arguments: an address representing the user and a uint representing the ID of the contract instance. The function retrieves this contract instance from the *contract_record* mapping and the role of the user from the *user_info* mapping. If the user is a landlord, it checks that they are indeed the landlord specified in this contract instance and returns whether they have paid their deposit. If the user is a student, it checks that they are indeed the student specified in this contract instance and returns whether they have paid their deposit.

The **check_if_already_in_contract** function is a private function that allows the contract to check whether a specific student or room is already in another contract instance. It takes two arguments: an address representing the student and a uint representing the ID of the room. The function retrieves the array of contract IDs associated with this student from their entry in the *user_info* mapping and checks whether it is non-empty. It also checks whether the specified room is occupied according to its entry in the *rooms_record* mapping. If either of these checks returns true, it returns true to indicate that either the student or the room is already in another contract instance.

The **check_user_already_registered** function is a private function that allows the contract to check whether a specific user has already registered. It takes one argument: an address representing the user. The function retrieves this user's entry from the *user_info* mapping and returns its already_init field.

The **delete_contract_instance** function is a private function that allows the contract to delete a specific contract instance. It takes four arguments: two addresses representing both parties to the contract, a uint representing the ID of their contract instance, and a uint representing the ID of their room. The function deletes this contract instance from both the *contract_record* mapping and both parties' entries in their respective *user_info* mapping. Then a distinction is made. If the value passed to the function as the room id is 0, then the function takes it as a default value that identifies the case where the contract ends successfully, so the room does not get marked as free. Otherwise, the *occupied* field in the corresponding *room* struct will be set to *false*.

The contract also has a **fallback** and **receive** function. The **fallback** function is executed when the contract receives a transaction that does not match any of its defined functions. It simply emits a Log event with information about the transaction. The **receive** function is executed when the contract receives Ether without any data (i.e. when someone sends Ether to the contract's address without calling any of its functions). It simply emits a Log event with information about the transaction.

The **remove** function is an overloaded private function that allows the contract to remove an element from an array. The first version takes two arguments: a storage array of uint elements and a uint element to remove. The function searches the array for the specified element and, if it finds it, removes it by swapping it with the last element in the array and then calling the pop function to remove the last element. If

it does not find the specified element, it reverts the transaction with an error message.

The second version of **remove** takes two arguments: a storage array of address elements and an address element to remove. It works in the same way as the first version but with address elements instead of uint elements.

5.3 Zero Knowledge Proof

As part of our implementation of the zero-knowledge proof algorithm, we have developed two functions: the **prover** and the **verifier**. The **prover** function takes private inputs and generates information that can be shared publicly. The **verifier** function then uses this information and performs secure mathematical computations to determine if the **prover** is indeed the owner of the private piece of information. This allows us to verify a student's status without revealing any private information.

Mathematically, the student proves their knowledge of an elliptic curve discrete logarithm. The curve we use is Secp256k1 which is the one used by Ethereum for the generation of public keys. The security of our protocol is equivalent to the elliptic curve discrete logarithm problem, assuming we have a cryptographically secure hash function and random number generator.

We adapted our algorithm from a well-known zero-knowledge protocol [Kog19] that used modular arithmetic with a large prime number (mod q algebra). We chose to use elliptic curves instead because they allow us to use smaller numbers (and therefore fewer bits), and also because Ethereum's private and public keys are computed using elliptic curve group operations, making it easier to adapt our algorithm for use in different contexts. We also made our algorithm non-interactive because, in blockchain technology, every call is public (including private inputs) so by making our algorithm non-interactive, we can perform the computation of the **prover** on a local node and then send the resulting value publicly to the **verifier** function, who can then determine if the user is telling the truth.

$\mathbf{P}(g, x, h = g^x)$:

$r \xleftarrow{R} \mathbb{Z}_q$

$u \leftarrow g^r$

$c \leftarrow H(g, h, u)$

$z \leftarrow r + cx$

output $\pi = (u, c, z)$

$\mathbf{V}(g, h = g^x, \pi = (u, c, z))$:

$c \stackrel{?}{=} H(g, h, u)$

$g^z \stackrel{?}{=} u \cdot h^c$

Figure 1: Zero-Knowledge Proof scheme

5.3.1 Account Verification

With our implementation, a student can prove their status as a student without revealing any private information. This is important because homeowners need to know

if they can use special contracts for students. We assume the existence of a public database containing encrypted personal information about students (such as encrypted private passwords). Using our algorithm, the student can prove to the homeowner that they know a value that, when encrypted, generates a value in the public database, without disclosing the value itself.

We now explore the use of zero-knowledge proof for account verification. We assume the existence of a tool that allows a user to execute the **prover** function privately, without disclosing the input. The output can then be made public to allow anyone to verify its authenticity.

Using the zero-knowledge proof protocol described above, we have designed a process for verifying accounts. During registration, the owner of a private key corresponding to an address can prove ownership without disclosing the private key. The owner can use a simple tool to produce proof of possession and send it to anyone who wishes to verify it.

We have implemented this process in Solidity to demonstrate its functionality. It is crucial for the user to produce the proof in a private environment since the computation involves the use of unencrypted private keys.

This approach offers both scalability and flexibility. It can be used for membership checks of any kind, such as verifying student status as described at the beginning of this section. The possibilities are limitless. Additionally, the elliptic curve can be changed for different use cases, providing further flexibility.

Overall, using zero-knowledge proof for account verification offers a secure and versatile solution for proving ownership without compromising privacy.

5.3.2 Description of the code

A more in-depth description of this part of the code will be given.

It consists of four main parts: a library called **Secp256k1**, a library called **EllipticCurve**, a library called **zkp**, and a contract called **accountVerification**. These files interact with each other to provide functionality for generating and verifying zero-knowledge proofs of ownership of an Ethereum address.

The **Secp256k1** library contains all the parameters of the elliptic curve Secp256k1, given in the form $y^2 = x^3 + ax + b$ defined over F_p. It also includes **getter** functions for each of these constants, allowing other contracts to access their values.

The **EllipticCurve** library contains all the functions to perform the operation between points of the curve: **ecAdd**, **ecSub**, and **ecMul** to perform, in order, addition and subtraction between two points and multiplication of a point by a scalar.

The **zkp** library implements a special type of zero-knowledge-proof using the elliptic curve Secp256k1 for operations. It includes three functions: **Prover**, **Verifier**, and **test**.

1. **Prover**: This function takes a private value x as input, which can be considered a private key or any other private code, like a password, and yields a proof (a sextuple) for the ownership of the value x. The function uses the imported libraries **Secp256k1**, **EllipticCurve** to perform elliptic curve operations. It generates a random value r using the block difficulty and timestamp and then calculates two points on the Secp256k1 curve: (h_x, h_y) and (u_x, u_y). The first point is calculated by multiplying the base point of the curve by the private value x, while the second point is calculated by multiplying the base point by the random value r. The function then calculates a hash value c_{256} using the **RIPEMD160** hash function applied to the concatenation of the coordinates of the two points, as well as the coordinates of the base point. It calculates $cx = c_{256} * x \mod N$ where N is the order of the base point, the operation is performed using the function **mulmod**, which does the multiplication mod N avoiding overflow problems. It then performs some modular arithmetic to calculate the value $z = cx + r \mod N$. Finally, it returns the sextuple $(u_x, u_y, c_{256}, z, h_x, h_y)$ as the proof.

2. **Verifier**: This function takes as input the sextuple generated by the **Prover** function and an encrypted value with the key x. It performs several elliptic curve operations using the imported libraries to verify that the proof is valid. First, it calculates two points on the Secp256k1 curve: (h_{cx}, h_{cy}) and $(point1_x, point1_y)$. The first point is calculated by multiplying the point (h_x, h_y) by the value c, which is c_{256} that comes from the proof. The second point is calculated by multiplying the base point of the curve by the value z. The function then calculates another point $(point2_x, point2_y)$ by adding the points (u_x, u_y) and (h_{cx}, h_{cy}) from the proof. Finally, it checks that the value c from is equal to a hash value calculated using **RIPEMD160** applied to the concatenation of $(u_x, u_y, G_x, G_y, h_x, h_y)$, where G_x and G_y are the base point of the curve, and that the points $(point1_x, point1_y)$ and $(point2_x, point2_y)$ are equal. If both conditions are satisfied, it returns true, indicating that the proof is valid.

3. **test**: This function is used to test the library's functionalities. It takes a private value x as input, generates a proof using the **Prover** function, and then verifies it using the **Verifier** function. The function returns a boolean value indicating whether the verification was successful or not.

The **accountVerification** contract provides several functions for verifying the ownership of an Ethereum address using the imported libraries. It includes several functions for generating and verifying zero-knowledge proofs of ownership.
The **getPub_kFromPriv_k** function takes in a private key and returns the corresponding public key.
The **getAddressFromPub_k** function takes in a public key and returns the corresponding Ethereum address.
The **verify** function takes in a private key and an Ethereum address and returns a

boolean value indicating whether the private key corresponds to the given Ethereum address.

The contract also includes several functions for generating and verifying zero-knowledge proofs using the **zkp** library.
The **zkp_accountGen** function takes in a private key and generates a zero-knowledge proof using the **Prover** function from the **zkp** library.
The are two versions of the **zkp_accountVer** function: one that takes in a sextuple generated by the **Prover** function and returns a boolean value indicating whether the proof is valid and corresponds to the address of the person that called it and another that takes in a sextuple generated by the **Prover** function, an Ethereum address, and returns a boolean value indicating whether the proof is valid and corresponds to the given Ethereum address.

Finally, the contract includes several functions for testing its functionalities, including two versions of a **sanity_test** function that performs a sanity check to see if the computation of an Ethereum address from a private key is up to date, and a **test** function that takes in a private key and an Ethereum address and returns whether they are associated using the zero-knowledge proof generated by the **zkp_accountGen** function.

Overall, these files provide functionality for generating and verifying zero-knowledge proofs of ownership of an Ethereum address using elliptic curve operations on Secp256k1.

References

[Kog19] Dima Kogan. "Proofs of Knowledge, Schnorr's protocol, NIZK". In: *CS 355: Topics in Cryptography* (2019). URL: https://crypto.stanford.edu/cs355/19sp/lec5.pdf.