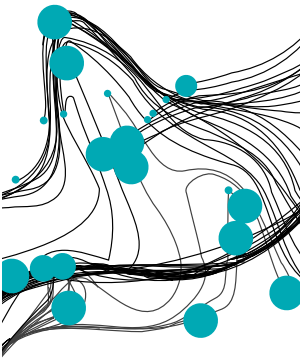
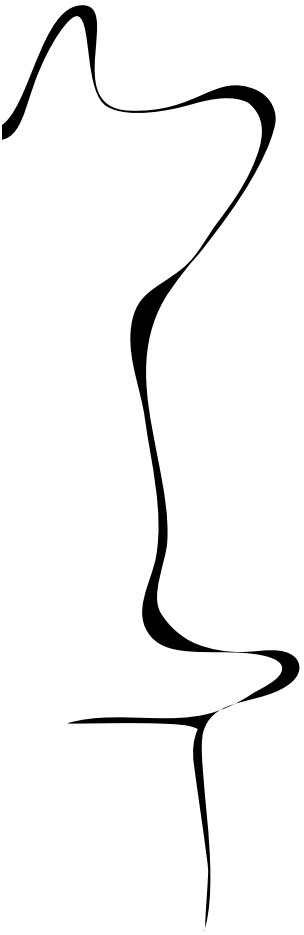


Privacy-Preserving Automated Rental Checking

Fedor Beets
Master Thesis
September 2018



Graduation Committee:
Dr. A. Peter
Dr. M. H. Everts
T.R. van de Kamp MSc
Services Cyber Security & Safety Group

Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Privacy-Preserving Automated Rental Checking

Fedor Beets*,
*University of Twente, Enschede, The Netherlands
Email: f.beets@student.utwente.nl

***Abstract*—Rental background checks ensure landlords only house good rentees, as rentees become hard to evict once renting a house. Rentees have to hand over sensitive information such as bank statements and marriage status to a landlord to rent an apartment. The landlord does not need that personal information, but only needs to know if the rentee meets certain requirements. If rentee information is stored forever and can be processed digitally, more stringent privacy requirements are needed. These background checks must be auditable so the landlord can reduce his liability, and automated so the checks can be done faster and tie into other systems. We merge the conflicting properties of preserving the rentees sensitive information while making the rental checks auditable.**

We define functional and privacy requirements for background checks on rentees. Using these requirements we create a scheme based around encrypted equality checks on a blockchain that is automated, auditable and privacy preserving.

A proof of concept implementation can check 19 different rentee properties on Ethereum for under 4 million gas or 4.93 euros. We now have a scheme that checks properties and makes these checks auditable, with a timestamp, and does not reveal the information that was checked.

I. Introduction

Depending on the competition for the apartment, the law and other factors, renting a new apartment means handing over a lot of personal information to a landlord. The landlord wants to know a history of previously rented apartments, marriage status, and bank transactions, according to the Dutch rental checking company Overbruggingsverhuur [1]. If rentees are rejected, they can hand over the information to the next landlord. These invasive checks are justified for the landlord because it is very hard to evict a rentee once the rentee moves in. Having a problematic rentee can mean damage to the apartment, not getting rent payments and can lead to complaints from neighbouring rentees.

On top of this, the landlord must by law check the income of a rentee for rent controlled apartments, and if the landlord performs their due diligence their liability is lower if anything goes wrong [2] [3]. For both the law and his liability, the landlord wants to be able to prove to an auditor later that he performed the necessary checks. Furthermore, current rental checks can take up to 5 days and cost 90 euros [4] [5] [6].

However, the landlord does not need all the bank transactions of a rentee. In general, the landlord only needs to know if the rentee passes the specific requirements for the apartment in question.

The problem is interesting because the requirements of privacy-preserving and auditability conflict. To make any solution auditable, information has to be recorded, stored and disclosed to third parties. Furthermore, not only does the landlord have to be convinced, the landlord must prove to third parties he is convinced.

We design a solution where the above mentioned sensitive information of the rentee is not revealed to the landlord, but only if the rentee meets the requirements. Because the information is not revealed, our solution preserves the privacy of the rentee. Moreover, our solution is automated, and not slower or more expensive than current rental background checks. The solution is also auditable, giving the landlord proof of due diligence.

To make our solution auditable, we use a blockchain. Because of a proof of work system, any alterations to a blockchain can be detected, assuming an honest majority [7]. By performing the background check using a blockchain, both the time of the check and the check itself are recorded and cannot be modified afterwards.

Because proofs have to be verifiable by a later auditor, we require a non-interactive solution. This is because interactive proofs cannot be verified by a third party without additional communication. Furthermore, using multiple rounds of communication on a blockchain takes a very long time due to messages having to be approved by the blockchain. We discuss this in more detail in section VIII.

For a rigorous treatment of the problem of making rental background checks automated, auditable and privacy-preserving, we create functional and privacy requirements for the problem. We designed and implemented a solution built on Ethereum using the Conjunctive Encrypted Equality Test scheme [8]. We test the performance of our proof of concept solution. We show that we can check 19 different properties of a rentee on Ethereum for under 4 million gas or 4.93 euros.

Our solution reduces the amount of information stored by the landlord. This means landlords cannot abuse their information about rentees, but also means they do not have to record the information and secure it, which reduces their risk. Our scheme is automated, providing a first step towards a much faster and cheaper rental checking process, which can then be used in more circumstances such as reducing risk in short-term lodging. By being auditable, we make the job of an auditor much easier, which can increase trust and reduce costs.

II. Background Knowledge

A. Pairings / Bilinear Map

The Conjunctive Encrypted Equality Test scheme uses pairings, also called bilinear maps. Take two additive cyclic groups

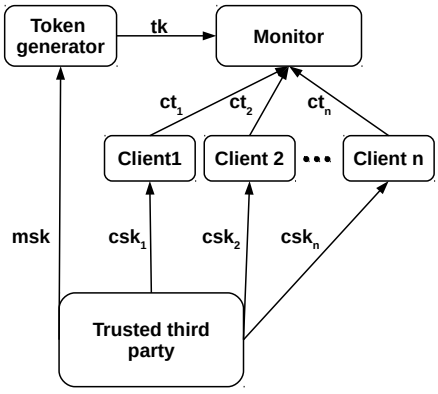


Fig. 1. Parties involved in CEET scheme as well as what messages are transferred between them.

$\mathbb{G}_1, \mathbb{G}_2$ of prime order q, with $\mathbb{G}_T$ another cyclic group written multiplicatively. A pairing is a map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, which has the properties of bilinearity and non-degeneracy.

- Bilinearity means for all $a, b \in \mathbb{Z}_p$, for all $X \in \mathbb{G}_1$, for all $Y \in \mathbb{G}_2 : e(aX, bY) = e(X, Y)^{ab}$.
- Non-degeneracy means there exist generators $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$ such that the order of $e(g_1, g_2) \in \mathbb{G}_T$ equals q, the order of $\mathbb{G}_T$.

We require a Type 3 pairing [9], where there is no efficiently computable homomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$. We use the function $\mathcal{G}(1^\kappa)$ to generate parameters for a Type 3 bilinear group for a security parameter κ.

B. Conjunctive Encrypted Equality Test Scheme

We use a *Conjunctive Encrypted Equality Test* (CEET) [8] scheme to evaluate the equality of two vectors. Several parties are involved in the CEET scheme. Involved are a trusted third party, a token generator, a monitor and several different clients.

The trusted third party generates and distributes secret keys, and does not learn the tokens. The token generator provides the token vector, but does not learn the aggregate vector. Several clients each contribute a small part of the aggregate vector. The monitor receives both vectors and checks if they are the same, but does not learn the values in the vectors. Figure 1 shows the parties and which messages they transfer.

A requirement of the CEET scheme is that the monitor should be able to check various combinations of token and aggregate vector for equality. But only given that they are encrypted under the same set of master secret keys and client secret keys. This means a token can be checked against multiple aggregate vectors. It should not be possible to take encrypted values from one aggregate vector and mix them with values from a different aggregate vector.

We give a formal description of the CEET scheme. We use $Z \xleftarrow{R} S$ to denote that Z is chosen as a uniform random element from the set S.

Setup(1^κ, n). Let $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, X, Y) \leftarrow \mathcal{G}(1^\kappa)$ be the parameters for a bilinear group. Where X,Y are the generators for the groups $\mathbb{G}_1, \mathbb{G}_2$ respectively. We have a pseudorandom permutation $\pi : \mathcal{K} \times \mathcal{M} \to \mathcal{M}$ for the message space $\mathcal{M} \subseteq \mathbb{Z}_p^*$. A cryptographic hash function $H_1 : \{0,1\}^* \to \mathbb{G}_1$. The parameters of the bilinear group and the two functions, π and H_1, make up the public parameters.

To generate keys, select $\gamma_i, \alpha_i \xleftarrow{R} \mathbb{Z}_p^*$ and $\beta_i \xleftarrow{R} \mathcal{K}$ for $1 \le i \le n$. The master secret key is

$$\text{msk} = \{(\alpha_i Y, \beta_i, \gamma_i Y)\}_{i=1}^n \ .$$

The secret encryption key for client i is

$$\text{csk}_i = (\alpha_i X, \beta_i, \gamma_i) \ .$$

Encrypt(csk$_i$, id, m_i). A client i can encrypt its message $m_i \in \mathcal{M}$ using csk$_i$ and an $r_i \xleftarrow{R} \mathbb{Z}_p^*$,

$$\text{ct}_i = (-H_1(\text{id}), -r_i X, \alpha_i \pi(\beta_i, m_i) r_i X + \gamma_i H_1(\text{id})) \ .$$

GenToken(msk, $\vec{y}$). The token generator can encrypt a vector $\vec{y} \in \mathcal{M}^n$ using msk. Choose $u_i \xleftarrow{R} \mathbb{Z}_p^*$ for $1 \le i \le n$,

$$\text{tk}_{\vec{y}} = \left(\{u_i Y, \alpha_i Y \pi(\beta_i, y_i) u_i\}_{1 \le i \le n}, \sum_{1 \le i \le n} (u_i \gamma_i Y) \right) \ .$$

Test(tk$_{\vec{y}}$, $\{ct_i\}_{1 \le i \le n}$). Output 1 if the following test is true, else output 0.

$$\prod_{1 \le i \le n} e(\alpha_i \pi(\beta_i, m_i) r_i X + \gamma_i H_1(\text{id}), u_i Y) \cdot \prod_{1 \le i \le n} e(-r_i X, \alpha_i \pi(\beta_i, y_i) u_i Y) e(-H_1(\text{id}), \sum_{1 \le i \le n} (u_i \gamma_i Y)) = 1 \ .$$

The CEET scheme is chosen-plaintext secure under the DDH assumption in group $\mathbb{G}_1$.

C. Ethereum

Ethereum is a blockchain-based distributed computing platform on which small programs, called smart contracts, can be run.

Smart contracts can be deployed by sending them to the network in a transaction, where the transactions are grouped into blocks. When a block is created by someone, then other nodes in the network verify its correctness. Once they verify it is correct, a smart contract is given an address and nodes include the effects of the transactions in the state of the blockchain.

The benefit of this distributed verification is that the proper execution of smart contracts is enforced by all nodes in the network. This means the contracts cannot be changed and are always executed as specified, and attacks such as double spending are prevented.

Unfortunately, distributed verification of computations also comes at a cost. Because transactions have to be repeated many times, any computation in a smart contract is much

more expensive than a computation done locally. Comparing Ethereum to Amazon EC2 cloud computing, it is 68 million times more expensive to add up numbers. Data storage is similarly very expensive.

The costs are based on the resources that the smart contract takes to run. This is implemented by assigning a cost to every low-level computational operation. These costs are deterministic, the same program always costs the same amount. The costs are paid for in what is called gas. There is an upper limit to the amount of computation that can be performed in a block. This limit is called the gas limit.

The currency of Ethereum is Ether, and not gas. However, Ether can be converted into gas through a bidding process. When a transaction is created it includes an exchange rate between ether and gas. Those who verify the blocks can accept this exchange rate and include the transaction, or reject it in favour of other transactions with more favourable rates. The price is based on the amount of transactions being submitted and is market-driven. If you set the exchange rate more favourably for those confirming, then your transaction will be processed faster. This separation of gas and Ether decouples computational cost and the price of Ether.

III. Functional Requirements

We want to construct an automated, auditable and privacy preserving solution for rental checking.

The landlord wants to verify a number of properties of the rentee automatically. The properties to be verified are a preset list, however, this list may change over longer periods of time.

We assume that the information needed to verify these properties rests with external parties such as credit reporting agencies or the government. One requirement is that there must be an automated method to request checks on these properties. Next, once the results of those checks are in, a decision should be made in an automated fashion. This decision happens according to preset specifications of what a rentee needs to adhere to.

Additionally, the landlord wishes to fulfil legal obligations. For example, obligations include not discriminating among rentees, not renting to rentees above a certain income for socialised housing. If the landlord is in violation of these obligations, he may be punished by the law. In contrast, under Dutch law the landlord can limit his liability in case the rentee uses the property for criminal purposes by proving that he performed background checks on rentees [2] [3]. Therefore, we require that records need to be kept of who rented what houses, starting when. This record needs to retain the information such as the income of rentees.

Another requirement is that the information should not be publicly accessible but upon request it must be disclosable by the landlord. For example, in the case of a government audit. The auditor should be able to view any information on the rentee, given the cooperation of the landlord. The last requirement of these records is that they must not be alterable.

Not all checks can be performed automatically. Employment is traditionally verified by a phone call, as other records may be faked. In such cases any partial verification possible is to be automated. Anything that can not be verified automatically should be fed into the decision process manually.

The results of these checks come in three formats. One example is Boolean results, coming from say an ID document which is either valid or invalid. Another example is categories, such as marriage status which falls into a few discrete categories. A last example is numerical values, for instance credit ratings come in ranges around 1-1000. These three examples encompass all types of results from checks. The system must be able to match based on Boolean values, the exact value of a category and provide smaller or greater than comparison for numerical values.

To accept a rentee, some properties may depend on other properties. For example, the total amount of debt is allowed to be higher if a person is married. Lastly a rentee should be able to check whether they have been accepted or find out the reason for why they were rejected.

To summarise, we have the following requirements for our solution:

1) The checks on rentee properties need to be automated.
2) Acceptation requirements change over time and must be changeable.
3) Acceptation requirements must be based on pre-existing guidelines on the rentee properties.
4) Records should be kept of who rented what, starting when.
5) Records should not be publicly accessible but be disclosable by landlord after the checking is finished.
6) Records should not be alterable by anyone.
7) All properties that can be verified by querying other organisations should be automatically decidable.
8) If a property must be manually verified, the property should still be usable as acceptation requirement.
9) Rentees should be acceptable based on comparison, category matching and specific true/false values.
10) The landlord must be able to decide what properties are checked.
11) Acceptation requirements for one property may be dependent on values of other properties.
12) Rentees should know on which aspects they were not accepted.

IV. Privacy Model

If data is stored digitally, then it is much easier to extract information from the data. Because we want to protect the rentees privacy, we need stringent privacy requirements for our *automated* rental checking scheme. To define the privacy requirements, we must first define how we expect the parties to act in our scheme.

A. Attacker model

First, there is the landlord who wants to rent out his house. We model the landlord as an honest but curious party, he does not take any malicious activities that can be detected by anyone. The landlord has a business reputation at stake and

does not want to cheat any prospective clients, but he stores any information he can use to improve his checking process.

The second party is the rentee, he is looking for a house to stay in and he would like to start living in the house owned by the landlord. We model the rentee party as malicious, he does not have any reputation at stake and may try to cheat the landlord to reach their goal of being accepted for the house.

The next party is the multiple different attestors who provide accurate information on the rentee. Examples of this may include government agencies who keep administrative information or commercial parties that share information as part of their revenue stream. Attestors are honest but curious parties whose business is selling correct information. Attestors may gather any additional information to build profiles of the landlord or rentee but cooperate honestly in any protocols.

Our fourth party is the *trusted third party* (TTP), who generates the requirements a rentee must adhere to. We model the TTP as an honest party, this is ensured by their public reputation which would be harmed if they take malicious actions and are detected. The TTP does not try to gather more information than they should according to the protocol. The TTP is only involved in setting up the requirements for the houses, and does not interact with the rentee or landlord.

The fifth party we name the public. We use a blockchain in our solution, the information on the blockchain is publicly available. The public is anyone who can read the blockchain, which includes everyone from nosy neighbours, competing landlords, intelligence services and anyone who looks up the publicly available information. What is important for this party is that a blockchain provides permanent storage, this makes any information disclosed to it permanently available to everyone. Due to the permanent and public properties of this party, the privacy requirements are very stringent, as any information can be dug up years afterwards when societal acceptances have changed. The blockchain is public, and the other parties can be a part of the public. As the rentee is modelled as malicious, the parties participating in the blockchain must also be modelled as malicious.

The sixth and last party is the auditor, for example a government organisation or the police. A party that ensures the landlord correctly followed the law or must retrieve information about the rentee under a court order. This party must be able to access all the information that the landlord has access to, as such we give them the same privacy requirement as the landlord and do not mention them in the rest of this section. This party is not involved in the main parts of the scheme but only afterwards. In principle an auditor is an honest party, but we construct a scheme that should still work if the auditor is willing to break protocol to learn more information, so we model the auditor as malicious. We sum all the parties and their modelling in table I.

TABLE I
ATTACKER MODELS: FOR EVERY PARTY, WHAT SECURITY MODEL DO WE EXPECT THEM TO ADHERE TO.

	Model
Landlord	Semi-Honest
Rentee	Malicious
TTP	Honest
Attestor	Semi-Honest
Public	Malicious
Auditor	Malicious

B. Privacy Requirements

Before we introduce information in the checking process, we introduce the notion of an attacker. This attacker is not given a single explicit role, but should instead be modelled as one of the actors of our scheme at a time. We now list descriptions of situations that could happen in the scheme, we do not yet judge whether the situations are allowed or should be prevented. No regard is given to the feasibility of learning the information, or any specific properties that could be checked.

1) The attacker learns information which significantly reduces the anonymity set of the [rentee/landlord/attestor/TTP]. For example their name.
2) The attackers finds out which properties the checking procedure verify. For example, what ranges of results are acceptable and mean success and which combinations of results result in rejection?
3) The attacker determines which house, uniquely identifiable by street address, is being rented out.
4) The attacker can determine the result of any individual check in a specific checking procedure. (e.g. 'is rentee A on a specific blacklist' 'yes/no').
5) Given two checking procedures the attacker can determine if the rentee is the same person or are different people. We call this linkability rentee.

To formalise these requirements and check their consistency, we translate the requirements into game-based privacy models. These are included in the appendix.

C. Who may learn what

We make a judgement for every piece of information, for every possible party, whether the party is allowed to learn that information. The results of these judgements are listed in table II.

Not all these privacy judgements are obvious, here we explain why the parties are not allowed to learn certain information. We explain them per party.

The attestors want to track the rentee across purchases to profile their behaviour, this can be monetised or abused. To prevent this attestors may not know the outcome of other checks. They may also not know which house is being rented out, or that the rentee is looking at other houses.

The TTP is picked by the landlord, and not the rentee. So the TTP does not need to know who the rentee is, which house is being rented out or what the properties of the rentee are.

Every party is part of the public, so if anyone is not allowed to know information, then the public may also not learn it.

TABLE II
PRIVACY REQUIREMENTS: FOR EVERY PARTY AND EVERY REQUIREMENT, EITHER THE PARTY MAY LEARN THE INFORMATION (Y) OR MAY NOT LEARN THE INFORMATION (N). BOLD ADDED FOR EMPHASIS.

	Rentee	Landlord	Attestors	TTP	Public
1.Id. Rentee	Y	Y	Y	**N**	**N**
1.Id. Landlord	Y	Y	Y	Y	**N**
1.Id. Attestor	Y	Y	Y	Y	Y
1.Id. TTP	Y	Y	Y	Y	Y
2.Which checks	Y	Y	**N**	Y	**N**
3.Which house	Y	Y	**N**	**N**	**N**
4.Outcome check	Y	**N**	Y	**N**	**N**
5.Linkability rentee	Y	**N**	**N**	Y	**N**

Additionally due to the permanent nature of the blockchain, and our scheme being tied to a specific rentee, the public may not know the identity of the landlord.

V. RENTEE REQUIREMENT EQUALITY SCHEME

This section introduces our proposed solution to the rental checking problem. First we introduce the algorithms used and then show how the algorithms together make up the scheme. Our scheme uses five algorithms, Setup, Encrypt, Test, CreateIdentifier, VerifyIdentifier.

We combine Setup and Gentoken from the CEET scheme into our Setup, though their functioning has not been changed. Encrypt and Test are identical to the CEET scheme.

Setup($\vec{y}$, 1^κ). Let n be the length of the vector $\vec{y}$. A trusted third party can take a vector $\vec{y}$ and output a token $\text{tk}_{\vec{y}}$, n secret encryption keys, csk and the hashing function H_2. Perform the Setup(1^κ, n) function of the CEET scheme as described in II-B. This gives the master secret key (msk) and secret encryption keys (csk_i) for $1 \leq i \leq n$. Next to the cryptographic hash function H_1 from the Setup of the CEET, we have a second cryptographic hash function $H_2 : (\{0,1\}^* \times \mathbb{Z}_p) \rightarrow \mathbb{Z}_p$. Perform the GenToken(msk, $\vec{y}$) function of the CEET scheme, giving $\text{tk}_{\vec{y}}$.

CreateIdentifier(PII, tid). A rentee can create an identifier id from some *Personally Identifying Information* (PII) $\in \{0,1\}^*$ and a transaction identifier tid $\in \mathbb{Z}_p$. The output is then,

$$\text{id} = H_2(\text{PII}, \text{tid}).$$

Encrypt(csk_i, id, m_i). A checking party can encrypt a message m_i using a secret encryption key for the checking party csk_i and an identifier id to create a ciphertext ct_i. Perform the Encrypt(csk_i, id, m_i) from the CEET scheme.

Test($\text{tk}_{\vec{y}}, \{\text{ct}_i\}_{1\leq i\leq n}$). Test lets the public compare whether a token $\text{tk}_{\vec{y}}$ and an aggregate vector $\{\text{ct}_i\}_{1\leq i\leq n}$ are equal. The result is the output of the Test($\text{tk}_{\vec{y}}, \{\text{ct}_i\}_{1\leq i\leq n}$) function of the CEET scheme.

VerifyIdentifier(PII, tid, ct_i). A landlord can check that the identifier used in the ciphertext ct_{id} is constructed correctly from the PII and the tid. The landlord checks that $-H_1(H_2(\text{PII}, \text{tid}))$ equals the value id in ct_i.

The scheme comes in two parts which are run after each other, first the setup is run and then after some time the rest of the scheme. There are two slight variations of the scheme, the flexible version and the store version of the scheme. We explain the flexible version in detail, and at the end show how the store version is different. All of the following steps are also depicted in figure 2 and figure 3.

We assume that before the scheme is run, there is agreement on a list of requirements that a rentee must fulfil. We run the scheme for a given apartment type, meaning a given list of requirements $\vec{y}$. The TTP takes the $\vec{y}$, selects a suitable 1^κ and performs Setup($\vec{y}$, 1^κ). The TTP distributes the csk's to the authorised attestors. The list of attestors and $\text{tk}_{\vec{y}}$ is then published by the TTP.

Some time later a rentee expresses interest in renting a specific house from the landlord. The landlord selects a $\text{tk}_{\vec{y}}$ that meets the requirements for the house he wants to rent out. Next, if the landlord already has a smart contract deployed containing the check, then he retrieves the location, and otherwise deploys the smart contract. The landlord sends a randomly generated tid, the location of the smart contract, the $\text{tk}_{\vec{y}}$ and the list of attestors to the rentee.

The rentee takes some PII, for example their name, and performs CreateIdentifier(PII, tid) to get an id. Two actions are then taken by the rentee. The rentee sends the id and the PII to the landlord. Additionally, the rentee also sends the id, the $\text{tk}_{\vec{y}}$ to each of the attestors in the list. The rentee authenticates himself and request that the attestor checks a property of the rentee.

Each attestor looks up which csk_i to use with this token. They look up the property they have on the rentee m_i and perform Encrypt(csk_i, id, m_i). The attestors then send the ct_i back to the rentee.

The rentee waits until he has all the ciphertexts from the attestors, and then calls the smart contract to perform the check, sending all ct_i's and $\text{tk}_{\vec{y}}$.

The blockchain then performs the test function Test($\text{tk}_{\vec{y}}, \{ct_i\}_{1\leq i\leq n}$) and the landlord can learn the result.

The landlord first verifies that the identifier is correct by performing VerifyIdentifier(PII, tid, ct_i). He obtains the $-H_1(\text{id})$ as part of the input to the Test function on the blockchain. If this fails the landlord rejects the rentee, if this

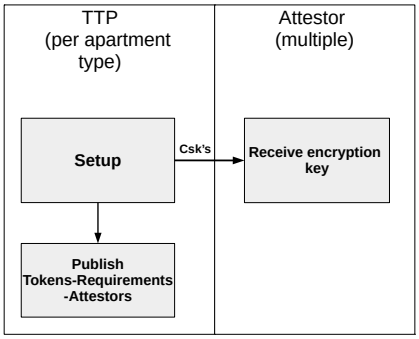


Fig. 2. Swimlane diagram of the first phase of our scheme, showing who performs what functions in what order.

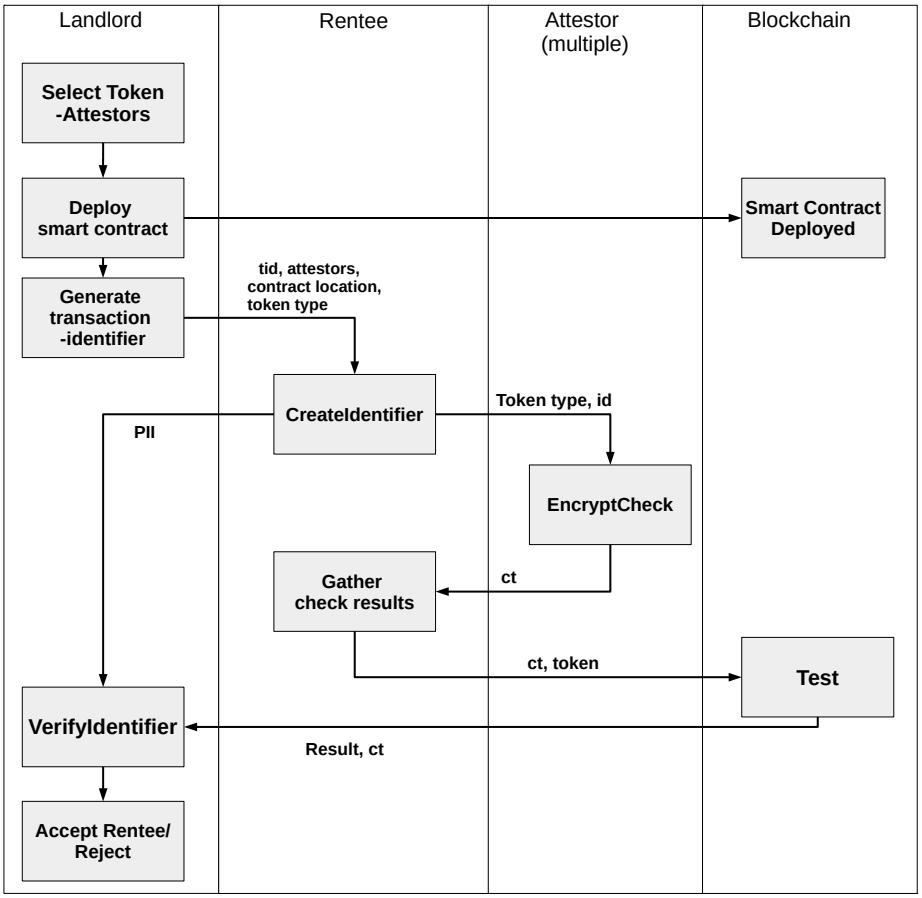


Fig. 3. Swimlane diagram of the main part of our scheme. Which parties perform which functions and what information is sent. The labels associated with the lines indicate information being sent, not all the input or output of the algorithms.

succeeds he checks the result of the test function. If the test function fails he rejects the rentee, if it succeeded then he accepts the rentee.

We call the scheme just described the flexible version of the scheme because the smart contract can be deployed once for every landlord, and then every token and list of properties can be tested on it. We also propose a second variation of this scheme, the store version. In the store version, the landlord creates a smart contract for each different token that is to be tested against, instead of a single contract that can handle all checks. In the store version, the contract can only test against a single token, and that token must be provided upon contract creation. Later the rentee still sends the encrypted check results to the smart contract, but no longer sends the token. By only sending the token once, the store version has to send less data per transaction which will lower the cost per test. We investigate the break even point at which the store version becomes cheaper than the flexible version.

A. Which Requirements Are Met

We explain how our scheme meets the requirements, and which requirement is not met. First we discuss the functional requirements.

Starting with the first requirement, our scheme is automated because the rentee properties can be automatically retrieved and decided on. To meet requirement two, the tokens can be changed to change acceptation requirements. On three, the tokens function as pre-existing guidelines. For the fourth functional requirement, we have the blockchain that records who starts renting when, including unfalsifiable timestamps.

Functional requirement five is met because the records are only identifying after the identifier is revealed. On requirement six, the blockchain is unalterable, given an honest majority, which makes the records unchangeable. On requirement seven, the properties can be automatically retrieved from attestors. Requirement eight is fulfilled because parts of the aggregate vector can be manually constructed.

The ninth requirement of matching based on comparison can be performed by matching against multiple tokens, in multiple tests, or by contracting all valid ranges into a single discrete category. The landlord can decide what token to check against, so requirement ten is met. Requirement eleven is met because properties may depend on the values of other properties inside a token. The twelfth requires additional effort, as the test

function does not show which parts of the token and aggregate vector are unequal. To meet requirement twelve the rentee must gather their information in plaintext from the attestors and compare it to the published list of tokens from the TTP.

Now, we discuss the privacy requirements, we reason out the first privacy requirement and use the game-based privacy models included in the appendix for the rest.

The first privacy requirement is met because we only publish the hashed PII of the rentee, meaning the public does not learn anything about the rentee. There is also no interaction between the rentee and the TTP, and so the TTP does not learn anything about the rentee. No information about the landlord is published on the blockchain, meaning the public does not learn anything about the landlord.

For the game-based privacy model of requirement two, the public has no additional information and cannot distinguish which rules or plaintexts are encrypted. The attestors, however, receive the encryption keys and can perform their own Encrypt function for two plaintexts that they sent in and thus learn the bit b. This means our scheme does not meet requirement two for the attestors, they can learn which ranges are acceptable. Attestors can then for example be bribed to change their reported values to acceptable ones. As the value of attestors is based on their reputation for honesty, this is not a major risk, however our scheme would be better if it did not violate this requirement.

Our scheme meets the third privacy requirement, because for neither the TTP, attestors or the public does a rental procedure reveal any information about the address of the house. The fourth privacy requirement is met because the landlord and the public have no information which lets them know which check results belong to which encrypted check results. The TTP could do this, because they have the encryption keys, however we model them as an honest party and assume they do not decrypt the check results.

The fifth requirement is also met. As we assume the landlord is not involved in both rental procedures, he does not know the identity of the rentee when he is not involved, and so cannot distinguish rentees. The attestors can similarly be prevented from following rentees, by using different attestors for different rental procedures, though if the same attestors are used repeatedly the rentee can be profiled. The public does not learn the identity of the rentee, and so cannot track him.

VI. Implementation

To show that the proposed scheme is feasible we built a proof-of-concept implementation[1]. Our system consists broadly of two parts, the smart contract and the off-chain parts. The off-chain part implements the Setup, Encrypt and a stub for CreateIdentifier, whereas the smart contract implements the Test algorithm. We have not implemented the VerifyIdentifier, as it does not effect the gas costs.

[1]See https://github.com/fedorbeets/RentalChecking

A. Smart Contract

We run our smart contract on the Ethereum blockchain, it implements our Test algorithm. The Test algorithm uses pairings, which are computationally complex. As the number of rentee properties (n) that are checked increase, the necessary amount of pairings grows as $2n + 1$. Because elliptic curve pairings are complex, we establish that sufficient pairings can be performed to support a realistic rental checking scheme. As the pairings happen on the Ethereum network, and must be done in single transaction, we must spend less gas than the gas limit. To demonstrate acceptable performance, our proof-of-concept implementation focuses on the test function of the scheme.

We chose Ethereum as our blockchain because it has precompiled contracts for the elliptic curve pairings we want to perform. Precompiled contracts are code that is executed outside the restricted Ethereum Virtual Machine, code executed in lower level languages which are faster. There are only 8 precompiles, the one for pairings performs pairings on a Barreto-Naehrig [10] curve with a curve order of 254 bits [11] and an embedding degree of 12. The pairings are the most computationally complex part of the scheme, and precompiles decrease the cost of the pairings as they are not restricted to the Ethereum Virtual Machines limited instruction set.

The smart contract is implemented in Solidity. At the lowest level of Ethereum computation is done in a stack-based bytecode language that is similar to assembly. Higher level languages have been created on top of the bytecode, the most widely used one being Solidity. Solidity can be compiled to bytecode by the Solidity compiler, and then the code can be run on Ethereum.

The Solidity smart contract we created implements the Test algorithm by calling the pairing precompile and reporting the results. As stated in section V, we create two versions of the test scheme, and so create two smart contracts, one for the flexible and one for the store version of our scheme. To reiterate, the Test algorithm is the only part of our scheme that runs on the Ethereum network, and so the only part that effects gas costs.

B. Off-Chain Implementation

The Setup, Encrypt and CreateIdentifier algorithms are run off-chain locally. Several details of these algorithms are left up to the implementation in our scheme description. These details are what algorithms to use for the pseudorandom permutation π, cryptographic elliptic hash function H_1, cryptographic hash function H_2, and what personally identifying information PII to use. The selection of the identifier does not effect the performance of the test function, so we used stub implementations for the PII, H_1 and H_2. The stubs for PII and H_2 means id $\xleftarrow{R} \mathbb{Z}_p$. Our stub for H_1 is not a secure hash function and should not be used in practice. Like the original CEET implementation, we use the pseudorandom function proposed by Naor and Reingold [12] as π, this function immediately maps the output onto the $\mathbb{G}_1$ elliptic curve.

We implemented the Setup, Encrypt and the stub for the CreateIdentifier in Python. These implemented algorithms use the same curve and underlying pairing library that are used by the Ethereum network to ensure compatibility.

Instead of performing tests on the Ethereum test network, we simulate a replica of the Ethereum network for development using Ganache [13]. Ganache simulates an exact replica of the Ethereum blockchain costs, and so we use Ganache for all our measurements.

VII. Evaluation

This section explains what was measured, what adjustments we made to the results of our measurements, then the setup and the results and lastly how the off-chain part of the scheme performs.

As explained in the previous section we focus on the gas costs of the equality tests. We measured gas costs for both the store and the flexible versions of the scheme from section V. The gas costs cover only the test algorithm of the scheme, as the Setup, EncryptCheck and CreateIdentifier are run off the blockchain. The computational cost of our scheme increases as more rentee properties are checked, so we measure gas costs by the number of properties checked, which we call token length.

A. Theoretical Evaluation

Ethereum has a different gas cost for non-zero input bytes and for input bytes that are zero. This makes our measured gas costs fluctuate. To make our results repeatable and to give an exact formula for the gas costs, we use the Ethereum Virtual Machine (EVM) specification to derive a worst case for our gas costs. This worst case comes in the form of two adjustments to our gas costs before we report them. Both adjustments involve the gas cost of input data to a transaction that calls the test algorithm.

Giving input data to a transaction costs gas according to the length of the input data. It is much cheaper to input a zero byte than to input any other value. This is because the Ethereum Virtual Machine is only addressable in chunks of 256 bits, meaning a Boolean true is 31 zero bytes followed by a 1. To prevent people from coming up with complicated schemes to avoid zero bytes, Ethereum efficiently compresses zero bytes and reduces their cost [14]. A zero byte costs 4 gas, whereas a non-zero byte costs 68 gas.

To explain the encoding of the input data [15], if an argument has a fixed length such as a boolean, then it is included in order. If an argument has a variable length, such as a dynamic array, then the location of the data is included instead, measured in bytes from the start of the arguments. At the location of the data, the length in 32-byte segments of the data is given, and then the actual data is included. We group the byte offsets and the data length under the term header.

The first adjustment is maximising gas cost of the actual data. Our data is points on elliptic curves, which has zero-bytes at a probability of roughly 1 in 225 [2]. We pretend there are no zero-bytes in the data, and increase the gas cost for every zero-byte in a transaction.

The second adjustment is making the cost of the header constant. The byte offsets deterministically cross a threshold such as 255 where it makes an additional byte non-zero. These thresholds are not linear, but happen at several token lengths, adding an extra gas cost. We calculate the gas costs for the header for each of our measured token lengths, and then take an average. The actual cost of the transaction is then adjusted as if the header had cost this average gas cost.

Both of these adjustments are minor in terms of gas cost, but as mentioned make the results repeatable and allow us to give an exact formula of the costs. The adjustments cause an average increase of 0.03% gas usage[3]. A vector with an average increase in cost is one of token length 9 in the flexible version, requiring 1,914,861 gas in one example, where we would report a maximised gas use of 1,915,437.

B. Results

The gas costs for the deployment and test functions of both the store and flexible versions are shown in figure 4. These numbers are after applying the two above mentioned adjustments.

To give a sense of scale, the maximum gas that a transaction can have (gas limit) has been just short of 8 million for the first 6 months of 2018. If you are willing to wait 30 minutes for confirmation, such a transaction would cost 0.016 ether, or 9.86 euros at the average exchange rate for the first 6 months of 2018 of 616 euros per Ether.

To interpret these results, our scheme can check 20 rentee properties for around 4 million gas, which makes our scheme usable for rental background checks.

C. Analysis Of Gas Costs

Now we can analyse why the results are not precisely linear after our two adjustments. As figure 4 shows, the gas costs are approximately linear in the token length. The exact formulas for the gas costs are given in table III. These formulas are after the two above mentioned adjustments are made. For small n, the gas costs are almost linear in the token length, but there is a small quadratic term. The small quadratic term comes from the cost of expanding the size of the memory.

Expanding the amount of memory used by a smart contract to store variables has a quadratic gas cost. Ethereum has a theoretically infinite memory, only limited by the gas cost. However, it costs increasing amounts of gas to address more memory. The memory cost function of Ethereum is $3a+\lfloor a^2/512 \rfloor$, where a is the size of the memory. The memory cost of an operation is the cost of memory before minus the cost of memory after. The quadratic component means it takes increasing amounts of gas to allocate new memory. This makes our gas costs quadratic, but for all realistic lengths the dominant terms are the base and the linear costs.

[2]Tested by sampling a million random points

[3]Tested by taking a single token and set of ciphertexts of each length between 1 and 10 for both the flexible and store versions.

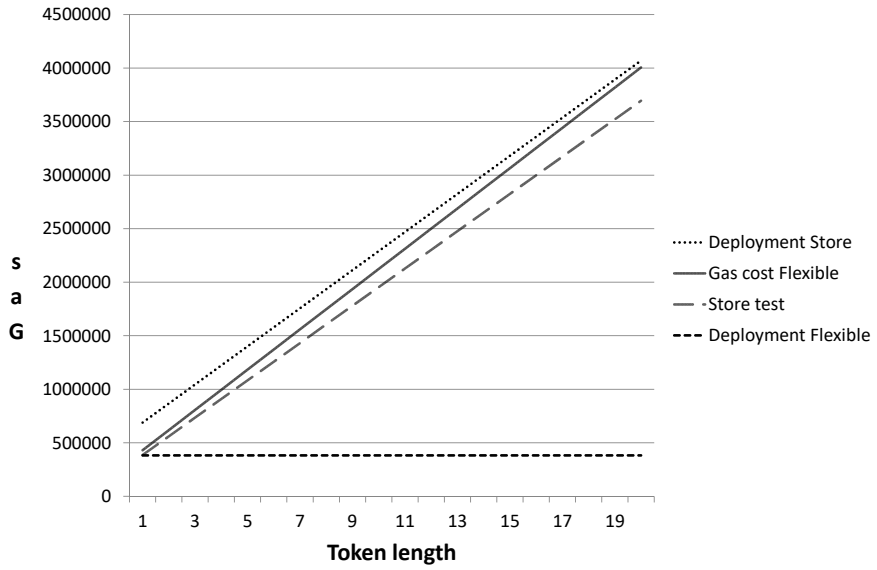


Fig. 4. Gas costs for both versions of our scheme, plotted against the amount of attributes being checked for (token length).

We locate the part of the code or Ethereum Virtual Machine assembly code that causes the memory to be expanded for further investigation. For the store and flexible equality tests, the expansion of the memory is caused by the re-ordering of the points on the curve from the input arguments to a form where they can be passed on to the precompile. The inputs are already present in the memory, however we know of no construct in the Solidity language to load them directly onto the stack in the correct ordering. They could be loaded in the correct order without memory expansion by the right Ethereum Virtual Machine assembly code, however that falls outside the scope of this paper.

There is an abnormality in the store version contract deployment, which has a separate base cost for the case where the token length is 1. This is due to instructions added by the Solidity compiler (for version 0.4.19), which as the last instructions before a return, loads some data from the contract and writes those into memory, then pushes bytes onto the stack and returns before using this memory. This happens equally in all versions of the store version contract deployment, but for the length 1 case it expands the size of the memory, incurring the extra 24 gas cost.

Examining the overall results, the store version is much more expensive to deploy but is cheaper per usage than the flexible version, as is expected. The difference in costs means that the store version of the scheme becomes cheaper in gas cost once it is used 13 or more times.

TABLE III
GAS COSTS PER VERSION.

Costs/Part	Base	per requirement length (n)
Deploy flexible	381,914	0
Flexible	241,618	$188{,}156n + \lfloor (n \cdot 24 + 25)^2/512 \rfloor$
Deploy store($n > 1$)	510,027	$178{,}016n + \lfloor (n \cdot 8 \ + 15)^2/512 \rfloor$
Deploy store($n = 1$)	510,051	$178{,}016n + \lfloor (n \cdot 8 \ + 23)^2/512 \rfloor$
Store	212,235	$174{,}124n + \lfloor (n \cdot 16 + 17)^2/512 \rfloor$

D. Off-chain Scheme Performance

We examine the run time of the set up, generation of the token and encryption of checks, to give a sense of how computationally intensive the scheme is. These are all the parts of the scheme that are run locally, and not on the Ethereum network. The run time is examined using cProfile Python profiler. The profiling is done on an laptop running a Intel Core i5-3340M CPU @ 2.70GHz.

We split our Setup into a SetupKeys part that generates the required keys and a GenToken part which generates a single token, along the lines of the original CEET scheme. We run the algorithms several times and take the fastest time for each token length up to 10 and fit a line through the data. Generating a token takes roughly 0.12 seconds plus 0.08 seconds per token length. The SetupKeys takes 0.08 seconds plus 0.07 seconds per token length. EncryptCheck takes 0.068 seconds for each individual check. This means that to keep each individual step under a second, we can use a token of length 10 and have GenToken take 0.92 seconds, SetupKeys take 0.78 seconds and each EncryptCheck taking 0.068 seconds. The SetupKeys

and GenToken are only run a few times by the TTP, whilst the SetupKeys, which is extremely fast, is run many times by the Attestors.

VIII. Related Work

Anonymous credentials were first proposed by Chaum [16], well known constructions include Idemix [17] and U-Prove [18]. Anonymous credentials allow a prover to selectively disclose information about their attributes to a verifier, without revealing the identity of the prover. The proofs in anonymous credentials are based on zero-knowledge proofs. In the rental checking problem, the attestors need to prove attributes to the landlord. Since acceptation requirements for one of these attributes may depend on the value of different attributes held by different attestors, our problem requires a proof over dependant values held by multiple provers. This can be solved by having a validator who takes in all the zero-knowledge proofs and then accepts or rejects the rentee. If the validator acts according to pre-defined rules and does not know the values of the zero-knowledge proofs, we have a description very close to the CEET scheme, except without a requirement for public verifiability. We know of no other non-interactive *multi-prover* zero-knowledge proofs where the values being proved may be dependant on one another.

Similar to anonymous credentials, secure multi-party computation can disclose information about attributes, without revealing the attributes [19] [20], this can also be made auditable [21]. We can use secure multi-party computation to compare a list of demands of a landlord to a list of attributes of the rentee. Because we can make the secure multi-party computation auditable, we do not need a blockchain for auditability, which also removes the need for non-interactivity. A second reason we use the blockchain is to provide a time when the checks were executed. If one can create a secure multi-party computation protocol that provides unfalsifiable proof of the timing of the checks, works with a malicious rentees, is auditable and performs fast enough then this would also be a solution to our problem. We did not look at this approach due to time constraints.

We can also classify a rentee as being a good rentee by using machine learning. Several companies [22] [23] are solving automated background checks in other fields. However, their techniques are trade secrets and not published, and they do not preserve the privacy of the party being checked. While most work is done on privacy-preserving training, few works look at privacy preserving classification [24]. The work on classification of encrypted data uses homomorphic encryption to support general machine learning models [24] [25] [26] or facial recognition specifically [27] [28]. However, this is in a setting where the server works on encrypted data, where the client decrypts the data to obtain the classification. The goal is to hide the model from the client, and the classification from the server. If the server has to learn the classification, then it must learn the inputs in plain text. As we want to hide the inputs from the server, we cannot make use of this work.

A different problem is that blockchains are public, and we want to log tests on the blockchain, but not reveal anything about the rentee's personal data. One system that deals with this problem is Enigma [29], which only stores hashes of data on the blockchain. We could store only a hash of the inputs and results on the blockchain, but this prevents automatically acting on the results on the blockchain, as the results need to be verified again. A different work on electric vehicle charging [30] uses a commitment based on a hash function which is logged to the blockchain, and the commitment is only ever revealed off-chain. This is the same approach that we use to hide the identifier of the CEET scheme. Hawk [31] looks related, but uses an off-chain manager who handles all user data and does computation on the data. This managing party must act correctly or it is financially penalised, but it is not prevented from disclosing or remembering user data. We are looking to prevent disclosure of the rentee data, and so cannot use the Hawk system.

IX. Discussion

A. How Much Does The Blockchain Add?

Running our scheme on the Ethereum blockchain increases the costs, what value does it add? Ethereum makes our scheme auditable, as a landlord is unable to alter the record of checks once made, and importantly it creates an unalterable timestamp of when the check occurred. Furthermore, by placing the check on the blockchain, the property can be set as rented out on a ledger of the landlords properties. Additionally, rent payments can be started automatically from the rentee. However, to not tie the rentee publicly to the performed checks privacy-friendly money would need to be used.

Ultimately, the rental housing industry currently does not use blockchains, and it still operates. As is done without blockchains, the landlord can keep an administrative record which a later auditor can examine, with trust in the ability of the auditor to detect forgeries. But by restricting himself to an unalterable administration, the honest landlord can create trust with the auditor.

B. Removing TTP And Committing Hashes

We looked at removing the TTP by using secure multi-party computation, however this option is too computationally expensive. The TTP performs the generation of checking keys and the generation of tokens. The original CEET scheme runs the token generation separately from the checking key generation so that the rules can be encrypted by a third party. We do not require the rules to be secret, so the tokens can be created between the landlord and the rentee. However if we reconstruct the tokens between the landlord and the rentee, then they have to be reconstructed for every rentee. The performance of the current scheme relies on only having to run the generation of tokens once. Removing the TTP would mean re-running the token generation for every new rentee using secure multi-party computation in a malicious setting, which would remove much of the performance of the scheme.

On a different note, a way to improve the performance of the scheme is to run the equality test portion of the scheme off the blockchain, and then only commit the inputs and the output to the blockchain. The equality test on Ethereum takes the most time and cost in the scheme, and running it off the blockchain increases the performance. If the costs of using a blockchain grow prohibitively expensive, then this is a valid option. However each party would have to take the inputs and verify the results for themselves. Not trusting the results on the blockchain immediately, prevents the results from automatically taking further action on the blockchain, such as being integrated into other systems for managing apartment rental on the blockchain.

C. What To Use As Identifier

Another point is that we have not yet fully specified what the requirements are for the PII the rentee uses to create the identifier, only that a name suffices. The only requirement is that the information distinguishes the rentee beyond reasonable doubt. The PII in the identifier commits the check to a person, so that no other person can later start renting the apartment and claim to have been checked before. The landlord cannot select a second rentee because they have the same name as the first, so a name is sufficient. To make this more identifying, a unique identifier such as the Burgerservicenummer or a uPort identifier [32] can be used. A uPort identifier uses Ethereum to record information about a person. A central registry proves the identifier is registered, and a user can add information to their identifier. This makes a uPort identifier a globally unique number that can establish someones identity.

D. Code Is Law Assumption

A different point is that a central assumption of this paper says that smart contract code is law, that we cannot modify the Ethereum blockchain. If this would not be the case the landlord could rewrite history and prove that a rentee had always been checked, rendering the blockchain useless. Whilst the Ethereum blockchain is largely free of modification there have been cases where the data has been altered that do not follow Ethereum protocol. The DAO incident [33] involved a bug in a contract that allowed a person to siphon 50 million dollars worth from the DAO to themselves. Whilst this was allowed by the code of the contract, it was obviously unintended. The result is that a large part of the community joined a fork of Ethereum where this money was restored [34]. This happened not by code in a contract, but by consensus of the community.

Recently there have been proposals to standardise recovery of lost Ethereum funds, for funds that have become unusable due to bugs in contracts [35]. This would put a committee in charge of how to reallocate funds, once a request for such lost funds is made. It would put a committee in charge of how to modify Ethereum without using the Ethereum protocol. The Ethereum blockchain, and any blockchain, can be modified, if the stakes are high enough. These are however highly documented changes, and for logging which rentee a landlord has checked the Ethereum blockchain is more than sufficient.

E. Further Work

Finally, what needs to be done before this system can be used in practice? Our system is a proof of concept, we did not focus on how landlords use background checks in practice. An inventory has be made of what criteria landlords currently use to approve rentees. How many different properties they use, and how inter-dependant these are. If there are too many combinations of valid attributes, then this could cause a combinatorial explosion in the options that need to be checked, making our scheme too expensive.

X. Conclusion

We have created a solution to the rental checking problem that is automated, auditable and privacy preserving. We made it auditable by performing the equality test on the Ethereum blockchain and making the identifier disclosable. The privacy of the rentee is preserved by only checking for equality on encrypted data through the CEET scheme. We formulated functional and privacy requirements, which solutions to the rental checking problem should fulfil. The scheme delivers acceptable performance, our proof of concept can run checks involving 19 variables for under 4 million gas or 4.93 euros. The off-chain part of the scheme also performs well, each step taking less than a second up to a token length of 10.

Using our solution, landlords store less information on rentees. Storing less information means landlords do not have to secure the information or pay for additional storage, but it also means landlords cannot abuse the information. Our scheme works towards a faster and cheaper rental checking process by being automated. Being faster and cheaper means background checks can also be used in more circumstances such as short-term lodging. The scheme being auditable means auditors can trust more of a landlords administration, which can reduce the costs of audits and improve the relationship between auditors and landlords.

References

[1] "Overbruggingsverhuur gegevens huurder (vacancy rental, rentee data)," http://overbruggingsverhuur.nl/uitleg_stap_3/gegevens_huurder.html, accessed: 2017-05-08.

[2] "Onderzoeksplicht bij verhuur? (research obligation on renting?)," http://hielkemaco.nl/nieuws/onderzoeksplicht-bij-verhuur/, accessed: 2018-06-28.

[3] "Opiumwet artiekel 11a (dutch law on opium, article 11a)," http://wetten.overheid.nl/jci1.3:c:BWBR0001941&artikel=11a&z=2018-04-27&g=2018-04-27, accessed: 2018-06-28.

[4] "Wonen-ok frequently asked questions," http://www.wonen-ok.nl/faq, accessed: 2017-06-16.

[5] "Leeghuis huurderscontrole (leeghuis rentee checking)," https://leeghuis.nl/2-algemeen/113-huurderscontrole.html, accessed: 2018-06-28.

[6] "Leeghuis: Onze aanpak en diensten (leeghuis: our approach and services)," https://www.controlehuurders.nl/diensten/, accessed: 2017-06-16.

[7] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[8] T. van de Kamp, A. Peter, M. H. Everts, and W. Jonker, "Multi-client predicate-only encryption for conjunctive equality tests," 2017.

[9] S. D. Galbraith, K. G. Paterson, and N. P. Smart, "Pairings for cryptographers," *Discrete Appl. Math.*, vol. 156, no. 16, pp. 3113–3121, Sep. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.dam.2007.12.010

[10] P. S. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *International Workshop on Selected Areas in Cryptography*. Springer, 2005, pp. 319–331.

[11] "Ethereum barreto-naehrig curve constants," https://github.com/ethereum/go-ethereum/blob/master/crypto/bn256/cloudflare/constants.go, accessed: 2018-08-01.

[12] M. Naor and O. Reingold, "Number-theoretic constructions of efficient pseudo-random functions," *J. ACM*, vol. 51, no. 2, pp. 231–262, Mar. 2004. [Online]. Available: http://doi.acm.org/10.1145/972639.972643

[13] "Ganache github," https://github.com/trufflesuite/ganache, accessed: 2018-03-07.

[14] "Design rationale: Gas and fees," https://github.com/ethereum/wiki/wiki/Design-Rationale, accessed: 2018-03-27.

[15] "Application binary interface specification," https://solidity.readthedocs.io/en/develop/abi-spec.html, accessed: 2018-03-27.

[16] D. Chaum, "Security without identification: Transaction systems to make big brother obsolete," *Communications of the ACM*, vol. 28, no. 10, pp. 1030–1044, 1985.

[17] J. Camenisch and E. Van Herreweghen, "Design and implementation of the idemix anonymous credential system," in *Proceedings of the 9th ACM conference on Computer and communications security*. ACM, 2002, pp. 21–30.

[18] C. Paquin and G. Zaverucha, "U-prove cryptographic specification v1. 1," *Technical Report, Microsoft Corporation*, 2011.

[19] G. Couteau, "New protocols for secure equality test and comparison," Cryptology ePrint Archive, Report 2016/544, 2016, https://eprint.iacr.org/2016/544.

[20] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 643–662.

[21] C. Baum, I. Damgård, and C. Orlandi, "Publicly auditable secure multi-party computation," in *International Conference on Security and Cryptography for Networks*. Springer, 2014, pp. 175–196.

[22] "Onfido," https://onfido.com/, accessed: 2017-06-27.

[23] "Checkr," https://checkr.com/, accessed: 2017-06-27.

[24] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, "Machine learning classification over encrypted data." in *NDSS*, vol. 4324, 2015, p. 4325.

[25] J. W. Bos, K. Lauter, and M. Naehrig, "Private predictive analysis on encrypted medical data," *Journal of biomedical informatics*, vol. 50, pp. 234–243, 2014.

[26] M. Barni, P. Failla, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider, "Privacy-preserving ecg classification with branching programs and neural networks," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 2, pp. 452–468, 2011.

[27] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft, "Privacy-preserving face recognition," in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2009, pp. 235–253.

[28] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "Efficient privacy-preserving face recognition," in *International Conference on Information Security and Cryptology*. Springer, 2009, pp. 229–244.

[29] G. Zyskind, O. Nathan, and A. Pentland, "Enigma: Decentralized computation platform with guaranteed privacy," *arXiv preprint arXiv:1506.03471*, 2015.

[30] F. Knirsch, A. Unterweger, and D. Engel, "Privacy-preserving blockchain-based electric vehicle charging with dynamic tariff decisions," *Computer Science-Research and Development*, vol. 33, no. 1-2, pp. 71–79, 2018.

[31] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 839–858.

[32] "uPort," https://www.uport.me, accessed: 2017-05-08.

[33] "The dao, the hack, the soft fork and the hard fork," https://www.cryptocompare.com/coins/guides/the-dao-the-hack-the-soft-fork-and-the-hard-fork/, accessed: 2018-06-12.

[34] "Hard fork completed," https://blog.ethereum.org/2016/07/20/hard-fork-completed/, accessed: 2018-06-12.

[35] "Standardized ethereum recovery proposals (erps)," https://github.com/ethereum/EIPs/pull/867, accessed: 2018-06-12.

APPENDIX

Security Games For Privacy Requirements

This section of the appendix presents a formalisation of the privacy requirements. We rephrased the privacy requirements set out in section IV in terms of advantage an adversary has over random guessing in security games.

Let the adversary be a probabilistic polynomial time algorithm. A renting procedure is a set of algorithms: *Setup, Createidentifier, VerifyIdentifier, Encrypt, Test*.

Which outputs and inputs are visible depend on the role of the adversary. This is analogous to the role of the attacker in the privacy requirements, corrupting a single actor at a time. Different roles receive different information from the challenger ($\mathcal{C}$) and may issue more inputs to algorithms. The following security games establish the definitions for the public role in the rental procedures. We do not create a security game for the first requirement, as it is sufficiently clear.

Requirement 2: We want to ensure that the public learns nothing about the predicates it is evaluating when it performs the test function.

Let Σ denote a finite set of plaintexts. Let F denote the set of all possible rules $f : \Sigma \to \{0, 1\}$.

We define *full security* as follows: given a set of plaintexts $x_1, x_2, \ldots, x_l$ and a set of rules $r_1, r_2, \ldots, r_m$ the adversary gains no information about the plaintexts or the rules other than the result of all the rules evaluating each plaintext. We establish full security through the following security game:

Setup: The challenger selects a random bit b.

Queries: $\mathcal{A}$ adaptively issues queries of one of the following types.

- Rule query: On the jth rule query $\mathcal{A}$ outputs two rules $\vec{R}_{0j}$ $\vec{R}_{1j}$. $\mathcal{C}$ performs **Setup**$(\vec{R}_b, 1^k)$ and remembers check keys $csk_{0j}, \ldots, csk_{nj}$ and responds $\mathrm{tk}_{\vec{y}b}$.
- Check query: $\mathcal{A}$ outputs two plaintexts P_0, P_1 consisting of results $P_{00}, \ldots, P_{0n}$ and $P_{10}, \ldots, P_{1n}$, $\mathcal{A}$ also outputs a number $l \leq j$. $\mathcal{C}$ generates a new identifier I and outputs **Encrypt**(csk_{il}, I, P_{bi}) for every i, and outputs I.

Assume an equality test function that tests if a plaintext matches a rule, like $f(P_m, R_m) \to 1 \iff P_{mn} = R_{mn}$ for every n and 0 otherwise. The adversaries queries have the restriction that for all checks P_0, P_1 and rules R_0, R_1 they must meet $f(P_0, R_0) = f(P_1, R_1)$.

Guess: $\mathcal{A}$ outputs a guess b' of b.

The advantage of the adversary is $|\Pr[b \stackrel{?}{=} b'] - \frac{1}{2}|$. The scheme meets the requirement if the advantage is less than ϵ for every adversary.

Requirement 3: An adversary is not allowed to learn what house a rental procedure is about, we define this to mean that an adversary may not learn what the physical address is of the house being rented out. We establish this through the following security game.

Setup: The challenger selects a random bit b.

Query phase 1: $\mathcal{A}$ outputs an address A_x. $\mathcal{C}$ simulates a

rental procedure the house at address A_x and responds $I, ct_1, \ldots, ct_n, tk_{\vec{y}}$ of that simulation. $\mathcal{A}$ may do this an arbitrary number of times.

Challenge phase: $\mathcal{A}$ outputs two addresses A_0, A_1 where the addresses are not equal. $\mathcal{C}$ simulates a rental procedure the house at address A_b and responds $I, ct_1, \ldots, ct_n, tk_{\vec{y}}$ of that simulation.

Query phase 2: The adversary may do the same as in query phase 1.

Guess: $\mathcal{A}$ outputs a guess b' of b.

The advantage of the adversary is $|\Pr[b \stackrel{?}{=} b'] - \frac{1}{2}|$. The scheme meets the requirement if the advantage is less than ϵ for every adversary.

Requirement 4:

We do not want the results of individual checks to become known to an adversary, this means that an adversary should be unable to distinguish even between two identical check results in separate procedures. We establish this through the following security game.

Setup: The challenger selects a random bit b.

Query phase 1: $\mathcal{A}$ outputs a check result m_x. $\mathcal{C}$ performs **Setup**$(\vec{y}, n)$ for a random $\vec{y}$, remembering csk of position x and generates a new identifier I. $\mathcal{C}$ responds **Encrypt**(csk_x, I, m_x). $\mathcal{A}$ may do this an arbitrary number of times.

Challenge phase: $\mathcal{A}$ outputs two check results, m_{x0}, m_{x1}. $\mathcal{C}$ performs **Setup**$(\vec{y}, n)$ for a random $\vec{y}$, remembering csk of position x and generates a new identifier I. $\mathcal{C}$ responds **Encrypt**(csk_x, I, m_{xb}).

Query phase 2: The adversary may do the same as in query phase 1.

Guess: $\mathcal{A}$ outputs a guess b' of b.

The advantage of the adversary is $|\Pr[b \stackrel{?}{=} b'] - \frac{1}{2}|$. The scheme meets the requirement if the advantage is less than ϵ for every adversary.

Requirement 5: Linkability rentee, the adversary may not determine if the rentee in two different checking procedures is the same person. See the following security game.

Setup: The challenger selects a random bit b.

Query phase 1: $\mathcal{A}$ selects a rentee R_x. $\mathcal{C}$ simulates a rental procedure involving rentee R_x and responds $I, ct_1, \ldots, ct_n, tk_{\vec{y}}$ of that simulation. $\mathcal{A}$ may do this an arbitrary number of times.

Challenge phase: $\mathcal{A}$ selects two rentees R_0, R_1 where the addresses are not equal. $\mathcal{C}$ simulates a rental procedure involving rentee R_b and responds $I, ct_1, \ldots, ct_n, tk_{\vec{y}}$ of that simulation.

Query phase 2: The adversary may do the same as in query phase 1.

Guess: $\mathcal{A}$ outputs a guess b' of b.

The advantage of the adversary is $|\Pr[b \stackrel{?}{=} b'] - \frac{1}{2}|$. The scheme meets the requirement if the advantage is less than ϵ for every adversary.