Alternative Authentication with Smart Contracts for Online Games

Michał Boroń
Faculty of Computing and Telecommunications
Poznan University of Technology
Poznań, Poland
Michal.Boron@cs.put.poznan.pl

Anna Kobusińska
Faculty of Computing and Telecommunications
Poznan University of Technology
Poznań, Poland
Anna.Kobusinska@cs.put.poznan.pl

2021 IEEE 46th Conference on Local Computer Networks (LCN) | 978-1-6654-1886-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/LCN52139.2021.9524886

Abstract—**Safely keeping the account's private key is vital in blockchain technology, as there are no native blockchain mechanisms for recovering the account when the private key is lost. Therefore, this paper proposes a solution to this problem, running directly on the blockchain platform in the form of a smart contract, enabling deposit of digital assets into the contract for safekeeping, along with a list of security questions. The proposed solution can be used, for example, in online games. By providing proof of knowing answers to questions, a player can access his account without his private key. Furthermore, additional security measures are put in place to cope with malicious behaviour.**

Index Terms—**Ethereum, smart contracts, online gaming**

I. Introduction

Recently, in the field of distributed systems, we have observed and experienced increased research activities, development and deployment of blockchain-based solutions [1]. Even though today blockchain technology is most commonly used in cryptocurrency systems, it also offers innovative solutions in other fields. One of them is the online gaming industry, where the information is permanently stored in immutable chain of blocks which can be used to verify and secure digital data such as game history, digital items and tokenized assets [2], [3].

Many popular blockchain-based online games are built on the Ethereum platform [4], on which, similarly to other popular blockchain platforms, the player's account is associated with a cryptographic key pair. The private key authenticates the player's account during interactions with blockchain by signing data sent to the blockchain network. As the public key of the account is known to the network, anybody (the game developer or other players) can verify that the player (wielder of the private key) requested signed data to be processed by the blockchain. Access to the private key is required to perform vital actions, such as transferring digital assets to another address, changing the state of the game, etc. Thus, losing the private key may have serious consequences. The distributed and open nature of permissionless blockchains comes with the limitation of being unable to 'reset' an account or forge a new private key for it. Therefore, when losing the private key, the account is practically impossible to recover, and the player loses access to his digital goods.

This research has been partially supported by the statutory funds of Poznan University of Technology

To preserve players privacy, it is desirable to develop a solution that does not involve a centralized approach. In this context, smart contracts [5], [6] seem to provide a suitable platform to build upon. Smart contracts are programs that run directly on the blockchain and perform actions when specific criteria are met (e.g. transferring digital assets when proof of shipment is presented). They do not involve any third party, enable distributed computation, and automated movement of digital assets on the blockchain.

This paper proposes a smart contract-based approach for recovering a player's account and the digital goods in the event of losing the private key. The proposed solution can run directly on the Ethereum platform, on which the online game is deployed, and does not require any additional external systems.

II. Concept

To enable access to digital assets stored in the blockchain after losing the private key, we propose an alternative authentication method. In order to preserve the decentralization, the solution does not rely on any trusted third party. The authentication is handled by a smart contract deployed on the blockchain. Player interacts with the contract via a client-side companion application installed on his device (PC, smartphone), which calls the contract on his behalf. Assets that are going to be accessible after completing alternative authentication have to be deposited into the smart contract. Those assets can be withdrawn from the contract either by issuing a transaction signed by player's account (similarly to any wallet contract) or using the alternative authentication method.

The alternative authentication method is based on security questions, i.e. player provides a set of several questions along with answers. Questions are stored in clear-text which makes it easier to remember associated answers, compared to using e.g. passwords. If one would prefer using passwords, appropriately modifying the contract would be straightforward. The difficulty of guessing the correct answers to security questions depends on the player who created them. As the player interacts with the contract via a client application, it is possible to enforce a level of complexity regarding the answers. Similarly to how many systems require passwords to contain a certain number of characters, including digits or special characters. Let $(iproof_{priv}, iproof_{pub})$ be a cryptographic key pair. Answers to the questions are used in symmetrical encryption algorithm to encrypt (and later decrypt) the private part of this key pair $iproof_{priv}$. Player proves his identity by signing with $iproof_{priv}$, which the contract verifies with $iproof_{pub}$ stored in the contract. Both data stored in the smart contract and operations executed in it are shared with all full nodes in the system. Thus, sensitive information, such as $iproof_{priv}$, cannot be processed by the smart contract in clear-text, or else it would be immediately compromised. That is why encryption occurs on the client-side, before data is passed to the smart contract.

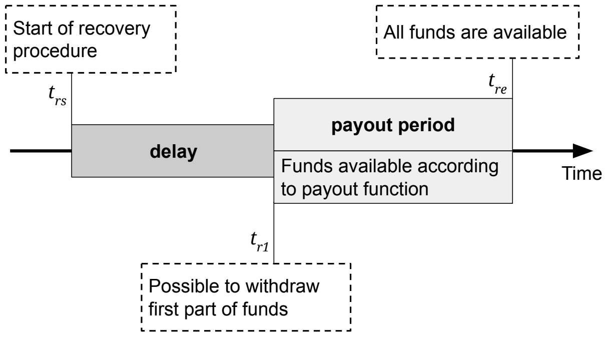
Algorithm 1 Client companion application: registering an account.

```
1: procedure REGISTRATION
2:     init(delay, duration, $iproof_{pub}$)
3:     apply a secret sharing algorithm to $iproof_{priv}$
4:     for all (Q, A) do
5:         addQuestion(Q, A(next share of $iproof_{priv}$))
6:     end for
7:     completeRegistration()
8: end procedure
```

Algorithm 2 Client companion application: recovering digital assets.

```
1: procedure RECOVERY
2:     create a new account $Acc_{new}$
3:     $i \leftarrow$ getQsLength()
4:     for $j \leftarrow 0; j < i; j \leftarrow j + 1$ do
5:         getQ(j)
6:     end for
7:     decode $iproof_{priv}$
8:     call startRecovery($Acc_{new}$ address) using $iproof$ account
9:     wait for delay period to pass
10:    (optional) cancelRecovery() using main account
11:    withdrawRecovery() using $Acc_{new}$, possibly multiple times
12: end procedure
```

The process of establishing the alternative authentication is summarized in Algorithm 1. Player generates a cryptographic key pair $(iproof_{priv}, iproof_{pub})$ and specifies the delay and duration parameters. Everything, except for $iproof_{priv}$, is passed to the contract via the `init` function. Then, $Q_1, ..., Q_n$ security questions and associated answers $A_1, ..., A_n$ are given by the player. The $iproof_{priv}$ is divided into n shares, using a secret sharing algorithm. Each share $S_i, for\ i \in (1...n)$ is encoded with the corresponding answer A_i. To prevent rainbow table attacks [7], it is necessary to encrypt with a salt such as the account's public key. Both division and encryption happens on client side. The text of security questions and



Start of recovery procedure
All funds are available
t_{rs}
t_{re}
delay
payout period
Funds available according to payout function
Time
t_{r1}
Possible to withdraw first part of funds

Fig. 1. The amount of digital assets available for withdrawal during the recovery procedure changes with time.

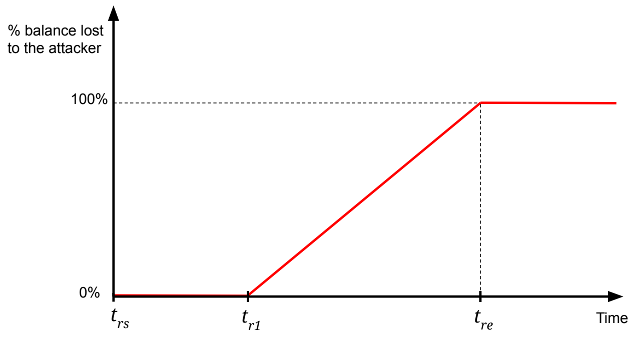


Fig. 2. The percentage of player's balance that can be withdrawn by an attacker who maliciously initiated a recovery procedure, depending on the moment when the player cancels that procedure. Assuming a linear payout function.

the encrypted shares of $iproof_{priv}$, that is $(Q_i, A_i(S_i))$, for $i \in (1...n)$ are stored in the smart contract. If, later on, the player wishes to change the security questions, the proof key pair has to be changed as well, to prevent an attacker from obtaining access by cracking the old security questions. Once registration is completed, it is possible to deposit and withdraw digital assets from the contract.

Steps that need to be taken in order to use alternative authentication are listed in Algorithm 2. Player creates a new account Acc_{new}, downloads security questions from the blockchain, answers them and reconstructs the $iproof_{priv}$ private key. He authorizes Acc_{new} account to access his assets by calling `startRecovery` function using iproof account and giving Acc_{new} address as argument. Afterwards, using the Acc_{new}, it is possible to apply for the digital assets. In order to prevent an attacker from immediately seizing all digital assets from an account upon cracking answers to security questions, the assets are released gradually over several days in a *recovery procedure*, as shown in Fig. 1 Smart contract allows Acc_{new} to make the first withdrawal after the delay period, whose length was specified earlier via `init` function, is over. When the delay ends, the recovery procedure enters the payout period. In this state, digital assets are gradually released according to a monotonous function (e.g. linear), called a *payout function*. The moment when the recovery procedure was started is denoted by t_{rs}, whereas the moments from the beginning to the end of the payout period are denoted by t_{ri}, where $i \in (1..e)$. The payout function takes as arguments the time elapsed from t_{r1} and the length of the duration period. It returns the percentage of digital assets that can be withdrawn at that time. When the payout period ends in t_{re}, all digital assets are available to Acc_{new}. The payout function can be any monotonous function, although it should be easy to compute since invoking it during the execution of code allowing the player to withdraw digital assets may require payment of an additional fee, proportional to the function complexity. To withdraw assets from the contract, player calls `withdrawRecovery` using Acc_{new} account. In order to determine which assets are to be transferred, the payout function is called internally.

The time window created by the delay state and payout period allows notifying the owner of the account, who is then able to request cancellation of the recovery procedure initiated by the malicious user. Concretely, the owner would call the `cancelRecovery` contract function from his main account Acc, proving that he still has access to his main account. An illustration of the effect of cancelling a maliciously initiated recovery procedure, for a linear payout function, at different moments in time is presented in Fig. 2.

A. Possible attacks

In the proposed solution, similarly to any authentication system, malicious actors may try to gain unauthorized access to the account to steal digital assets from players. To illustrate such behaviour, let us consider a situation where Alice registered her security questions and deposited some digital assets into the contract. As a result, a proof key pair $iproof_{priv}, iproof_{pub}$ was tied to her account. Erick is a malicious actor that will try to steal Alice's digital assets and move them to his account.

In the first step, Erick gets Alice's security questions either by calling the appropriate function in the smart contract or by directly inspecting the state of the smart contract on the blockchain (as smart contract state and data are persisted on the blockchain). Next, Erick tries to crack the security questions. As their strength depends on Alice, who invented them, the answers to security questions may be easy to guess. Nevertheless, due to using a secret sharing algorithm to encrypt pieces of $iproof_{priv}$, Erick can confirm that he has the correct answers only after obtaining all of them (by checking if the decoded private key matches $iproof_{pub}$, which is stored in the contract and publicly visible). If he obtains only part of the answers, then, by the properties of the secret-sharing algorithm, it does not give him any useful information to reconstruct the $iproof_{priv}$. Assuming that Erick managed to decode $iproof_{priv}$, he has two options as to how to proceed further. The first one is to initiate the recovery procedure, and the second one is to wait for Alice herself to do it after losing her private key (if that happens).

TABLE I
USER SCENARIOS.

Name	Deposit	Withdrawal	Recovery withdrawal
Player A	Every week	1 time	—
Player B	Every week	Every week	—
Player C	3 times	1 time	—
Recovery procedure	Never	Never	1 time

In the first scenario, Erick initiates the recovery procedure for Alice's account. Alice can be notified of the initiation of the recovery procedure by the smart contract, making it easier to react to this event. As shown in Fig. 2, if Alice reacts in time, her digital assets are safe. However, the more time elapses without her reaction, the larger portion of digital assets can be taken by Erick. The exact value of stolen digital assets depends on the chosen payout function specifying the rate at which digital assets are released.

Erick could decide to wait until Alice really lost Acc private key instead of initiating the recovery procedure himself. This would ensure that Alice would not be able to cancel the recovery procedure. However, Erick would have to be the first to present data signed by $iproof_{priv}$. To fulfill these criteria, Erick would have to intercept and block (or at least delay) Alice's transaction before it is embedded in a block. This kind of attack is not feasible.

As a smart contract cannot protect encrypted data from being seen by an attacker, it is impossible to throttle the number of retries to guess the right answers. Also, the attacker can try to answer any number of questions that seem to have an obvious answer (or a limited domain of possible answers). Furthermore, as account balances are public knowledge, an attacker can easily target only the players owning large amounts of digital assets.

III. Cost evaluation

The proposed solution was analyzed concerning the monetary costs of using the contract by performing simulations. Simulation for use cases corresponding to different user profiles (patterns of behaviour) was run with the help of Truffle Suite for JavaScript. The proposed system comprises two parts: a smart contract deployed on the blockchain and a client-side companion application. The contract was written in the Solidity version 0.6.X. To determine the cost of a particular transaction for a given gas price, it was multiplied by the value of cumulative gas used, obtained from `web3.eth.getTransaction()`.

Table I shows parameters used in scenarios simulating behaviours of different players. A player can interact with the contract by sending money into the contract for safekeeping (**deposit**) or pulling the digital assets back from the contract (**withdrawal**). Furthermore, a player can initiate the recovery procedure and gradually withdraw the digital assets (**recovery withdrawal**). The recovery procedure is considered in isolation, that is, without the preceding deposit or withdrawal operations, to clearly distinguish between the costs related to recovery and those associated with maintaining an account.

TABLE II
YEARLY TRANSACTION COSTS FOR PLAYERS A, B, C AND TRANSACTION COSTS OF 1 WITHDRAWAL RECOVERY PROCESS.

Player A				
Gas	**Transaction Costs**			
[Gwei]	[Finney]	[€ 1:85]	[€ 1:158]	[€ 1:280]
1	1.474332	0.13	0.23	0.41
3	4.422996	0.38	0.70	1.24
20	29.48664	2.51	4.66	8.26
40	58.97328	5.01	9.32	16.51
60	88.45992	7.52	13.98	24.77

Player B				
Gas	**Transaction Costs**			
[Gwei]	[Finney]	[€ 1:85]	[€ 1:158]	[€ 1:280]
1	3.259568	0.28	0.52	0.91
3	9.778704	0.83	1.55	2.74
20	65.19136	5.54	10.30	18.25
40	130.38272	11.08	20.60	36.51
60	195.57408	16.62	30.90	54.76

Player C				
Gas	**Transaction Costs**			
[Gwei]	[Finney]	[€ 1:85]	[€ 1:158]	[€ 1:280]
1	0.117516	0.01	0.02	0.03
3	0.352548	0.03	0.06	0.10
20	2.35032	0.20	0.37	0.66
40	4.70064	0.40	0.74	1.32
60	7.05096	0.60	1.11	1.97

1 withdrawal recovery procedure				
Gas	**Transaction Costs**			
[Gwei]	[Finney]	[€ 1:85]	[€ 1:158]	[€ 1:280]
1	0.08866	0.01	0.01	0.02
3	0.26598	0.02	0.04	0.07
20	1.7732	0.15	0.28	0.50
40	3.5464	0.30	0.56	0.99
60	5.3196	0.45	0.84	1.49

Each scenario considered a period of player activity of 12 months (52 weeks) and a linear payout function that unlocks new portions of digital assets daily.

Results for each player are shown in Table II. In Ethereum, computing each smart contract instruction has a fee, denoted by *gas price*. Since gas price depends on many factors, such as how quickly the user wants to process the transaction or how much other users are proposing to the miners, the results are shown for multiple values of the gas price. Gas price is expressed in units of gwei, equivalent to 10^{-9}ETH. It ranges from 1 gwei (very low but currently transactions with this gas price are successfully processed), 3 (satisfactory but slow), 20 (normal), 40 (high), 60 (very high). The other dimension of the table features the costs of all transactions in a given user scenario for a given currency or currency conversion ratio. Concretely, the cost of all transactions in Finney (1/100 of ETH) and the value in € currency, based on varying conversion rates, observed in recent years. The value in Euro currency gives an intuition regarding the order of magnitude of the values involved.

To conclude, we chose two data points: an average case (gas price of 20 gwei and conversion [€ 1:158]) and the most pessimistic case (maximum gas price and the most expensive conversion rate). As mentioned previously, the costs are calculated for the period of 1 year. Player A, who was quite active, with one deposit per week and one withdrawal, had to spend an average of 4.66€ and in the worst-case 24.77€. Player B issued the most transactions, i.e. two per week, and had to pay 10.30€on average, and 54.76€ in the worst case. Player C with only 4 transactions spent on average 0.37€, and 1.97€ in the worst case. The recovery procedure costed 0.28€ on average and 1.49€ in the worst case. The results show that frequent interactions with the contract, especially depositing and withdrawing, can generate substantial costs in the range of few tens of euro. However, the cost of a recovery procedure is rather low at a few euros. Thus, the best way to use the contract would be to deposit digital assets, which will not be needed in the immediate future. Then the deposit/withdrawal costs are under control, and the player is still protected from losing assets. It is hard to evaluate the solution's affordability, as concrete amounts in the euro heavily depend on gas price and conversion rate. However, even if one were to cautiously assume the pessimistic costs for frequent interaction with the contract, which amount to 20-50 euro per year, they are still within reason. Furthermore, moderately frequent interaction with the contract, like player C, brings the costs down to the range of 0.37-1.50€ which is very cheap. In conclusion, usage costs are satisfactory, especially considering the limitations brought by the decentralization assumptions.

IV. Conclusions

This paper introduced a mechanism that supports the online game player to deal with losing the private key to his blockchain account. The solution took the form of a smart contract. Analysis of the costs of using the application, in gwei and euro, revealed that the costs are reasonable, especially considering the limits brought by the decentralization assumptions. In future work, we plan to address the issue of the attacker being able to target wealthy users based on the relation between security questions of an account and digital assets belonging to it, by subjecting that relation to obfuscation.

References

[1] J. Yli-Huumo, D. Ko, S. Choi, S. Park, and K. Smolander, "Where is current research on blockchain technology?—a systematic review," *PloS one*, vol. 11, no. 10, p. e0163477, 2016.

[2] A. Pfeiffer, S. Kriglstein, and T. Wernbacher, "Blockchain technologies and games: A proper match?" in *FDG '20: International Conference on the Foundations of Digital Games, Bugibba, Malta, September 15-18, 2020*. ACM, 2020, pp. 71:1–71:4.

[3] T. Min, H. Wang, Y. Guo, and W. Cai, "Blockchain games: A survey," in *IEEE Conference on Games, CoG 2019, London, United Kingdom, August 20-23, 2019*. IEEE, 2019, pp. 1–8.

[4] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[5] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, 2014.

[6] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 254–269.

[7] G. Brose, "Rainbow tables," in *Encyclopedia of Cryptography and Security, 2nd Ed*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, pp. 1021–1022.