Annales Mathematicae et Informaticae
57 (2023) pp. 1–23
DOI: https://doi.org/10.33039/ami.2023.07.002
URL: https://ami.uni-eszterhazy.hu

# Blockchain diploma authenticity verification system using smart contract technology

**Ruben Frisch, Dóra Éva Dobák, József Udvaros**

Budapest Business School,Faculty of Finance and Accountancy,
Department of Business Information Technology
frischruben1998@gmail.com
dobak.dora@uni-bge.hu
udvaros.jozsef@uni-bge.hu

**Abstract.** Blockchain technology and smart contracts have huge potential which has not been exploited fully yet. The main objective of this paper is to showcase the powerful attributes of blockchain technology and smart contracts, showcasing our unique and powerful use case of document verification in the field of higher education using the Ethereum protocol. Our smart contract use case will take advantage of the main attributes of blockchain technology to solve the problem of document forgery. These amazing attributes are immutability, censorship resistance, extreme robusticity, transparency, and neutrality, in addition to near-perfect availability and decentralization. Ethereum enables developers to create decentralized applications without having to invest in expensive infrastructure. Document forgery has a very long track record in the education sector and academia. In this digital age, it has become frighteningly simple and inexpensive to acquire fake university diplomas, certificates, and many other types of credentials. This has a long-term negative effect on higher-level education because it damages the healthy competitive environment of students and the reputation and credibility of institutions. The most problematic version of the diploma which is the most susceptible to forgery is physical diplomas. Even with relatively expensive and difficult-to-replicate security elements, such as holograms and special security markings, these are not efficient enough to keep bad actors away from trying to forge them and replicate them. The more complex methods we use for preventing physical document forgery, the more knowledge and experience does the verifier needs beforehand due to the complexity and the unique nature of anti-forgery methods and materials one has to look for dur-

Submitted: August 10, 2022
Accepted: July 14, 2023
Published online: July 17, 2023

ing examination. The verification process of continually evolving and changing physical forgery prevention stamps, materials, holograms and others are expensive to automate the verification procedure and introduces additional human labor cost (training staff, hiring new employees, hiring trainers and forgery specialists). Therefore the best prevention method for replication is to build a system that makes it infeasible to even try to commit forgery. Usually, when an employer asks for the diploma, the student sends an electronic photocopy of the document to them or scans it. This completely nullifies the effect of holograms, watermarks, special UV active materials, and all other physical security elements. Even with the use of centralized electronic document verification systems, data manipulation is still possible, in addition, such a system introduces the concept of having to trust a third party for verification and single point of failure, in addition to lack of transparency, immutability, and data availability. An ideal solution would be one that is trustless, transparent, immutable, and always accessible. Blockchain technology offers the optimal solution to document forgery. In this article, we will showcase our Ethereum smart contract solution and all of the crucial aspects of document integrity.

*Keywords:* Smart contract, blockchain, diploma

*AMS Subject Classification:* 94-06

# 1. Introduction

Document forgery poses a great risk to the reputation and credibility of the academic field. Counterfeit diplomas and certificates damage higher education greatly. We aimed to develop a blockchain-based smart contract solution, which will help in battling diploma mills and forgery services by registering documents into a secure blockchain environment. Employers and institutions wish to verify documents securely and in a quick and simple way. Registration, and verification process especially should not be a time and human resource-consuming process, and should not require high-level skills to distinguish fake and real documents. In our smart contract use case, we developed our contract to be able to verify the authenticity of any type of document, as long as it has unique and descriptive data attributes, which could be used to generate a fingerprint of the document using hash algorithms such as SHA-256 [18]. We mainly focus on the field of higher education, where document forgery, especially in the case of diplomas and certificates is very common and is still a huge concern. Blockchain is a perfect solution, because of its decentralized and immutable nature, where network participants can monitor every transaction on-chain and verify data themselves without having to trust an intermediary or third party to verify and store information [3].

This smart contract implementation takes advantage of the security guarantees of the Ethereum network [2]. It should be noted that this particular smart contract could be used for any kind of document verification, as long as the document has a unique fingerprint, which is calculated from a primary key, in addition to other descriptive type data elements, such as date, name, grade, and many others hashed

together to make a single, unique and fixed-length data.

One of the main expectations towards our smart contract design is data privacy. Documents contain sensitive information which should not be placed on the blockchain in raw text format. That is why the data should be always hashed first before broadcasting a document registration transaction [20]. Hashing the data makes it nearly impossible to decrypt, it is a one-way encryption function. This data security aspect is especially important in the case of blockchains, where data cannot be removed afterward, it is immutable. The content of valid blocks cannot be modified after they become part of the chain, this is one of the major fundamental attributes of blockchains that guarantees data immutability and data integrity. Data immutability is extremely important in our use case of document verification, that is why there exists no better technology than a public decentralized blockchain with amazing security guarantees in place.

Without secure hashing functions, this use case would be impossible to implement optimally. A hash function generates fixed-length data from variable-length data inputs, making it ideal to create a unique fingerprint of the document at hand. It is also crucial that one should not be able to recover the input data from the hash itself. A hash function should always generate the same output for the same input. If a user has possession of the document's data elements, the user should have the ability to generate the hash from the data. With the hash, the user can query the contract's state with a pre-defined query function, which tells the user if the document has been registered on the blockchain or not, returning with a logical value of true or false. So the hash function must be deterministic for this use case to work as intended. Even if one makes a small error in the input data, a completely different hash will be generated.

One other extremely important attribute of hash functions is collision resistance, two documents should never have the same hash representation. In theory, hash collision is possible with an extremely low chance, so before broadcasting a document registration function into the network, the company or university should verify that the hash does not already exist in the database. If it does, changing the document's primary key or any data component should solve the issue, but this is again an extremely rare scenario. With long enough hash outputs, preferably 256 bits or more, the possibility of collision becomes almost impossible. Although it should be nearly impossible to calculate the input data from the hash, due to its one-way nature, it should also be fairly easy to generate the hash from the raw input data. A hash function is needed that is fast and efficient, but secure at the same time. An SHA-256 or SHA-512 hash function for example would be ideal for our use case. A hash function must have a pre-defined range in our smart contract design, meaning that the output of the hash algorithm shall have a fixed length regardless of the input size [16].

The smart contract must have an owner, which by default will be the EOA (Externally Owned Account) type Ethereum account that initiates the registration of the smart contract with the special contract creation transaction. The smart contract constructor runs one time, specifically when the contract is created. The

constructor will set the owner's address as the owner of the smart contract who will have special privileges, such as document registration. The owner is saved in a state variable, which is an address type variable or object. We must also implement a function that will take care of owner changing, although this function could be discarded based on the specific requirements of the institution or company. Specific functions of the contract will require the function caller to be the contract's owner, this rule will be enforced by a function modifier, which is always activated when the function is called. The function modifier has the job to decide if the function caller has the same address as the smart contract's defined owner stored in the state variable.

The query function used to verify document authenticity is enabled to call for any user, which is the whole point of this application. Anyone can verify the document by having the necessary data at hand by calling a function. The document verification function is free to call because functions that do not change the state of the contract do not require a transaction that changes the state of the blockchain by including the transaction in a block. On the other hand, document registration and many other functions that write or modify the storage of the contract will cost gas and require a signed and broadcasted transaction on the network [5]. In the case of universities, it is important to have a function that enables mass document registration, which is also implemented in our smart contract code. This will make it much easier to manage and register diplomas for universities, not having to do it one by one. There is also an option for mass document queries to make the mass verification process faster and easier to manage, although this function is rarely useful in most scenarios. When designing a smart contract, we must think forward, because there is no way to change the code of the contract later. It should have more functionality than the minimum requirement just in case it will be needed in the future.

The source code of the smart contract should always be made publicly available. This is usually done by adding the source code to an Ethereum block explorer. The most popular block explorer by far is called Etherscan, which has the function to verify source code. The code is compiled into bytecode, then the service compares the result of the compilation with the actual bytecode of the already registered smart contract bytecode. If they match, then the source code is verified and made public. Without this step, only the bytecode format of the source code is available for users on the Ethereum network, which is hard to read for most users and developers alike. The smart contract code is immutable, so after we register the contract on the blockchain, it is impossible to modify or add new functions. Thankfully there are Ethereum test networks for this reason, so projects can test the smart contract meticulously before registering it on the main network. The best test networks are Kovan and Ropsten, these have almost identical properties like the main network, with the most notable exception being gas prices. Gas prices differ between the test network and the main network because of significant differences in network usage.

During the development phase of the solution, we used Solidity smart contract

programming language, which is a high-level, object-oriented language best suited for contract development. The version we used to code the smart contract is Solidity with solc compiler version 0.8.7. The integrated development environment we chose is Remix IDE, which has all the tools needed to develop efficiently for this fairly straightforward use case. Remix also has advanced text manipulation features, in addition to source code compiler to bytecode format and contract creation transaction automation. Therefore the IDE handles all the required actions such as creating and broadcasting the contract creation transaction and compiling Solidity source code into bytecode [21].

## 1.1. Introduction of the main issue with traditional document verification models

In our digital world, data quality and authenticity have utmost importance and focus. Centralized information systems are often not transparent enough and always require trust [22]. Manipulation and forgery of documents is a huge issue in the field of higher education, where diplomas and certificates are used as a signaling mechanism of one's acquired knowledge and skillset. Physical copies can be easily forged, but digitally signed PDF diplomas are not ideal either. One possible route is a centralized document verification system [1]. Centralized services lack the extreme security guarantees that Ethereum has as a public blockchain, such attributes for example are availability, immutability, and transparency [10]. Another less than optimal way to battle forgery is to make physical copies difficult to counterfeit. Although this is not an ideal solution, because it is extremely expensive, requires special materials, technology, and machines to produce. Physical copies can also be lost by the owner or even stolen, can be damaged by water and fire, and degrades with time. On the blockchain, data is always available, is immutable after it has been included in a valid block, and every transaction is visible to all users of the network [6]. Another problem that exists today is that diplomas and other sensitive documents are too hard to be verified. In our smart contract based solution, the verification process is very simple and efficient and can be done by anyone who has the required data at hand. Another issue with centralized document verification systems is the strong dependence on the institution's hardware and software infrastructure [11].

Outsourcing such sensitive tasks is a huge risk too. Data can be easily modified by the institution later, the documents are not immutable, there is a risk of corruption or human mistake. Trusting a third party is not ideal, especially when the document has such high value. There is also the high cost of centralized systems, they often require expensive infrastructure and have a significant maintenance cost. Hardware failure is another risk, which in the case of a decentralized blockchain database is mitigated by all full nodes having a copy of the state of the blockchain. When broadcasting information to a blockchain, we must be very careful. The data will stay on the blockchain forever, meaning that there is no room for error when registering documents. The institution must implement security measures to mitigate these risks. Such risk could be a hash collision or inaccurate document

information. Before submitting data into the smart contract, the documents must go through a strict review phase, where syntax and semantics are checked.

# 2. Methodologies and methods

## 2.1. Analysis of viability and practicability

The implementation of our blockchain document verification system is fairly easy and straightforward. Positive data redundancy plays an important role in this system, the institution is advised to store all the registered documents in a secure database server. Remember, that only the document hash would be registered on the blockchain, the document itself and all of its information resides in the relational database. An optional component would be a web-based user interface, to make document queries more user-friendly, although modern block explorers already have smart contract function calling capabilities, such as Etherscan, which enables manual interaction with the contract itself. It is also good to note, that as every single transaction is permanent and visible to all network participants on the blockchain, one could query through all the transactions where the contract address was the recipient, and eventually find the document hash manually, without even calling the document query function.

In regards to the practicability of using public blockchain and smart contract technology, this design has many advantages over a centralized solution [9]. The hardware infrastructure needs of our implementation are lower than in the case of centralized solutions, due to the fact that the document hashes are stored on the blockchain directly in the forms of immutable and censorship-resistant transactions. The smart contract's storage is secured by the Ethereum protocol and its many nodes. Centralized solutions require trust from the owners of the documents. In the case of blockchain documents, there is no way to modify or delete registered documents. In truth, after registering a document with a valid transaction, the transaction will be always contained in the corresponding block, thus modifying the storage state of the smart contract will not truly remove the document from the blockchain, as transactions are immutable after being included on the chain. Even if the university or company ceases to exist for any reason, the registered documents will stay forever on the blockchain, making it a timeless and superior solution.

The contract is programmed in a way that document duplication is impossible, although hash collisions shall be checked strictly off-chain to prevent anomalies. The document fingerprint hash should be passed on to the correct query function in the smart contract to verify that the particular hash has not been already registered by the institution. Another check would be trying to match the generated hash with a hash already stored in the relational database the institution maintains for document data storage. Such collisions are extremely rare when using the correct hashing algorithm, but this use case is extremely sensitive as we are talking about mostly certificates and diplomas where errors are unforgivable. Humans are

prone to errors, multiple off-chain assertation mechanisms should be implemented to prevent incorrect document registrations with the smart contract.

The document registration process is simple and easy to manage for the administrator. First, the document hash must be generated, then checked if it already exists in the database or not. If not, then the contract owner shall call the registration function of the smart contract, and pass the document hash as the argument. The transaction must be signed before broadcasting it to the network with the correct private key, which is only known to the owner of the contract. If the private key is lost, the system is compromised. The contract has a function that enables the owner to pass ownership of the contract to another address. Regardless of the inclusion of this function, losing the private key or getting it compromised is always catastrophic. After the transaction is signed, it can be broadcasted to the network. Usage of a secure hardware wallet is crucial and a basic security best practice, where the private key is always isolated in a secure element encryption hardware component. The hardware wallet can sign transactions without ever revealing the private key or putting the private key data into the random access memory of the computer. Even if the computer is infected, the private key won't be accessible, it is encrypted and isolated on the hardware wallet, even when signing a transaction. The transaction must be broadcasted with the correct amount of ether as a transaction fee. The transaction fee amount varies, it must be always checked before sending the transaction to the network. In case of setting the transaction fee too low, it will not be mined at all. To solve this issue, the owner should send the same signed transaction again, but with the same nonce value as the original one, so that the original will be overwritten by the new transaction with the correct transaction fee set. It is good to note, that transactions can be stuck forever if the gas price is not set high enough, so it is advised to set it higher than the minimum value. Stuck transactions can still be corrected by broadcasting another transaction with the same nonce as the original.

A valid transaction, which is included in a block is usually considered final by convention when there are six additional blocks placed on the containing block. Ethereum blocks are created and placed on the chain about every 12-13 seconds on average, which means that a document registration transaction on average takes about 72-78 seconds to be considered extremely immutable. Theoretically, even if block production is stopped momentarily, the transaction cannot be modified after it is placed on the chain, because that would require more than half of the consensus nodes to agree on that false state of truth. The block production time varies based on many factors, such as dynamic difficulty set by the consensus mechanism. The consensus mechanism controls block production pace by setting the difficulty dynamically. It tries to maintain that 12-13 seconds for successful block mining to maintain security and stability. Lowering the block production time is dangerous for the network, as it can increase hardware requirements for the mining nodes, as faster block time needs quicker synchronization and more powerful hardware. The more expensive hardware is needed, the fewer nodes will be on the network due to a higher barrier of entry. Fewer nodes mean less decentralization, which results

in decreased robustness and security. Another aspect of too short time duration between successful blocks is the fact that 51% attack becomes easier to conduct, as calculating block identifier hashes become much less time and energy-consuming. Bitcoin has an average of 10 minutes block production time, which would be way too slow for a smart contract platform like Ethereum. On a smart contract enabled blockchain, the number of broadcasted transactions is significantly higher, thus it requires faster block production time to keep up with the block space demand of network users.

After the document hash has been successfully registered by the owner of the contract, there is no additional step. The document owner should receive the document in digital format along with the calculated hash. When the graduated student wishes to prove to the employer that he or she has the required certificate or diploma for the job application, then the student should send an email or chat message to the employer with the document data along with the hash. Then the employer can go to the document verification website, fill out the form quickly, then the hash is generated. The hash is passed as an argument when calling the read type function of the smart contract, which has a logical returning value of false or true. If the hash has been already registered, the function returns true, the employer has successfully verified that the applicant has the necessary document for the job application. Another verification route would be for the employer to calculate the hash manually, then use some kind of block explorer or wallet to interact with the smart contract.

The most decentralized and secure, yet quite time-consuming verification process would be to check the smart contract transaction history in a block explorer and see if the document hash has been registered or not based on transactional evidence. Having an archive-type full node would be the most secure way, by running your own Ethereum node, with client software compiled by yourself from the source code. Although these methods are bothersome and require too much background knowledge to be a feasible alternative to using a pre-built front-end or block explorer. Of course, in case the document owner passes false document data and a correct hash, the verifier might make a mistake and not calculate the hash him or herself. That is why the verifier should always check if the hash generated from the document data is valid, as the output is deterministic. Sending only the document data without the hash is fine too, that way this kind of manipulation attempt is mitigated by having the verifier generate the document fingerprint.

## 2.2. Document integrity and permission levels

It is crucial to determine in a decentralized application, who is able to call specific functions of the smart contract. Some functions might only be able to be called upon by the owner of the contract, while others may be called by everyone using the Ethereum network and signing, then broadcasting a valid transaction. In the case of diplomas and certificates, only the company that has the right to emit these documents should be able to register such documents into the smart contract and blockchain. This is why we needed to implement the smart contract in a way that

there is an explicitly defined owner of the contract with special permissions, such as document registration, ownership transfer, and contract condition configuration. It is also important to mention that the contract implementation should be optimized for the specific requirements of the company or institution, therefore some functions might be excluded or new ones might have to be included in the code, aligning to the given specification.

In our smart contract implementation, there is only one owner, who has the ability to call upon all existing functions of the contract and control it to the full extent. More permission levels could be implemented, such as multiple owners, multi-signature document registration, and sub-owners. Although introducing more levels and actors to the system might increase the gas cost of the transactions, due to the fact that these changes will always result in more complex code, which is more expensive to execute for the EVM (Ethereum Virtual Machine). The owner of the smart contract should always be the one who emits the documents, for example in the case of diplomas this would be the university itself. The owner of the contract can be easily determined by either monitoring the specific address type variable of the contract, or by calling the query function which will return with the address of the owner. The company or institution should also make the owner's address public, in addition to the valid smart contract address to avoid confusion and remain fully transparent.

Keeping the owner's private key safe is of utmost importance. Some might believe that this task is easy and self-evident. Making sure that the private key never gets stolen or leaked is a difficult task, which requires a safety mechanism to be set and executed properly. The owner of the smart contract should always use a trustworthy hardware wallet or some kind of enclave technology to keep the private key completely isolated and encrypted at all times, even when signing a transaction. It should never be copied into the random access memory of the machine either in a raw format, as this opens up new possibilities for private key leakage. A multi-signature implementation of transaction signing would significantly improve the security of the contract, although it makes the document registration process more time-consuming, in addition to increasing gas costs. The multi-signature would be generated by two or more owners, meaning that it would always require a minimum of two separate transactions to sign and broadcast a document registration or other, which is of course more expensive than having only one owner who controls everything about the contract. There are always drawbacks of a given mechanism, therefore the contract should be coded with the specific requirements in mind.

In the case of universities, the multi-signature implementation might be preferred due to its increased security guarantees. The cost of the transactions could be mitigated by implementing a mass document registration function, which would take an array of document fingerprint hashes as an argument. This way, there is no need to generate the multi-owner signature for each and every document at hand, only one signature is required to make the mass registration possible. The contract would have a variable for each owner, which stores the outcome of their signature logically. Each owner would make a special sign transaction, which would modify

the bool variable to the outcome of the transaction. If all of the corresponding sign condition variables are true, then the registration function could be called and executed properly. If one of the owners would not sign it, the registration call would fail and revert. There are quite a lot of ways to implement a multi-signature mechanism, there are many aspects to consider such as cost efficiency, data security, code complexity, permission levels, institution or company-specific requirements, legal environment, and more.

Smart contracts are immutable, meaning that after registering a contract on the blockchain none shall change its code or address. Smart contracts also have a state, which is determined by their storage memory. The storage memory consists of variables, objects, and primitive data types. By implementing a special method called *selfdestruct*, the caller can destroy the smart contract's code and storage memory, rendering it empty so to say. An important aspect of self-destructing lies in its implementation explicitness, which means that a contract only has the ability to self-destruct if it is hardcoded into the contract. Therefore it is always up to the given specifications and requirements if the contract should have a self-destruct function or not. The main use case of such a method is testing contracts, finding bugs in deployed contracts, then self-destructing it when the contract is no longer needed. It also has an important part in the contract migration mechanism, where the contract left behind gets self-destructed after the migration is complete. Another important aspect is the fact that after the completion of self-destruction, the contract's address and transaction history remains untouched, therefore a contract cannot be purged completely, the address and transactions will always remain, only the code and storage is destroyed.

In our implementation, we use an on-off switch kind of smart contract condition mechanism, in which the contract's owner can call a specific contract activation and deactivation function. In case the contract needs to be shut down temporarily, it can easily be done, without purging the code and storage of the contract like in the case of self-destruction. Almost every function of the contract shuts down in case of deactivation, although document query remains active at all times, meaning that everyone will be able to verify documents even after shutting down the contract. The owner passes a bool value of true or false to the corresponding function, then the value is assigned to the bool contract state variable. After that, calling functions will be impossible, except for some specific methods. The condition check is done by a function modifier structure when calling methods. After each call, the contract checks if the contract is turned off or on, and proceeds to execute on the correct path according to the state of the contract. Even though there is an on-off switch built-in, the smart contract is still fully decentralized, participants can still verify documents as usual. The implementation is of course optional, it could be excluded from the code or changed accordingly to the needs of the company or institution.

## 2.3. Cost efficiency

The document verification system has a decentralized on-chain part in the form of the smart contract, which stores document hashes and other state variable values,

and has the required logic in form of Solidity code. Even though smart contracts do not require maintenance in a traditional sense, it still has costs in form of the transaction fees when calling write type functions. Write type functions make changes to the internal state of the smart contract, to the storage memory of it. The storage memory is stored on the blockchain where the smart contract's every component resides. The contract address, code, storage, and transactions are all stored on the blockchain. When you call a function that adds or modifies the state of the contract, the transaction will cost you a non-deterministic amount of gas fee. The transaction fee is non-deterministic because it is unknown which code path will get executed when calling a function, it depends on the argument of the function call, the time the transaction has been broadcasted, and many more factors that can take a role. Of course, it is possible to calculate the estimated gas units you will need based on empirical data, but the system itself cannot determine it with certainty. Each transaction gas unit will cost a non-deterministic amount of ether in the form of Wei. Miners will only add transactions to the blockchain which have a sufficient amount of gas included with the transaction, otherwise, it will be ignored by the vast majority of the miners and the transaction might get stuck forever. That is why the owner of the smart contract will always have to check the current gas prices to avoid issues with transaction finalization. It is advised to pay higher fees in order to ensure that the transaction will be included in a block for sure. Other factors, such as the time of transaction propagation could also be crucial to minimize transaction fees because the gas price tends to fluctuate greatly based on day times. The cost of gas units is mostly affected by the network's capacity utilization, the number of transactions competing for block space, and blockchain inclusion [17].

On the other hand, read-type functions do not require any gas to be paid, as they only read from the contract's storage memory. Read-only type functions are often marked as *view* or *pure* in the declaration, meaning that they will never attempt to modify or add data to the storage of the contract, only read from it. As all of the document verification functions are read-only, they will not cost anything for users to call them. Another cost of smart contracts is the contract creation special transaction, this one is always needed in order to set up the smart contract on the blockchain [15].

**Algorithm 1.** Smart contract Solidity source code events, storage state attributes, constructor, fallback special function and modifiers.

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.7.0 <0.9.0;

contract DocumentVerificationContract {

    //Events
    event documentRegistrationEvent(
        address transactionSender,
        bytes32 documentHash,
```

```
        uint epochSeconds,
        uint blockHeight
    );

    event massDocumentRegistrationEvent(
        address transactionSender,
        bytes32[] arrayOfDocumentHashes,
        uint epochSeconds,
        uint blockHeight
    );

    event setOwnerAddressEvent(
        address transactionSender,
        address newOwnerAddress,
        uint epochSeconds,
        uint blockHeight
    );

    event fallbackEvent(
        address transactionSender,
        string fallbackMessage,
        uint epochSeconds,
        uint blockHeight
    );

    event setContractStateEvent(
        address transactionSender,
        bool stateChangedTo,
        uint epochSeconds,
        uint blockHeight
    );

    //Storage state attributes
    bool private contractState = true;

    address private contractOwner;

    mapping(bytes32 => bool) private documentMapping;

    bytes32[] private documentHashes;

    //Constructor
    constructor() {
        contractOwner = msg.sender;
    }

    //Fallback special function
    fallback() external {
        emit fallbackEvent(
            msg.sender,
            "fallback method activated: "
            "Wrong function prototype or empty ether call",
            block.timestamp,
            block.number
        );
    }
```

```
67
68      //Modifiers
69      modifier onlyOwner() {
70          require(msg.sender == contractOwner);
71          _;
72      }
73
74      modifier contractStateIsActivated() {
75          require(contractState);
76          _;
77      }
78
79      //...
80      //Function code and detailed explanations in the upcoming sections.
81  }
```

## 2.4. More about data privacy and security

The different kinds of documents that are suited for this use case often contain sensitive, private information which must be kept hidden. Blockchain transactions are irreversible as soon as they are included in a block and the block is valid, therefore one must be extremely careful when submitting information in the forms of transactions. In this document verification implementation, we must never put raw, private data on the blockchain. Meanwhile, users must be able to verify and prove the authenticity of documents in case they possess the correct data series to calculate the unique fingerprint of the document.

A perfect solution is using hashing algorithms, which have attributes fitting perfectly for this use case. This way reversing the hash into the correct data structure is extremely time-consuming for an attacker, a near impossible task. Therefore a document should always have some kind of primary key that is unique, this key might be composed of several descriptive attributes. The document's primary key should have a decent length and should be composed of letters and numbers. Usage of sub hashes is also a viable option, although it would make hash generation more resource-consuming and the complexity of the system would increase somewhat.

A well-designed hash function is always deterministic, passing the same arguments to it will always generate the same output. Another crucial attribute of such hash functions is collision resistance, every different input data should be mapped to different output hash values. Two different documents should never have the same hash fingerprint, but in extremely rare cases it could still happen. That is why a collision check should be implemented off-chain to prevent such issues before broadcasting a document registration transaction. The document hash should be calculated almost effortlessly, consuming minimal hardware resources. Also, the hash function should always return with fixed length hashes, in the case of SHA-256 that is 256 bits. A good hash function makes hashes that are almost impossible to reverse, and are always deterministic with the input data. From a data security perspective and general efficiency and design, using hashes for our use case is optimal [13].

## 2.5. Smart contract ownership implementation

The owner of the contract is stored in the address data type variable called *contractOwner*. It is set as *private*, therefore only the containing contract can see it. Setting variables to *private* visibility do not mean that they cannot be monitored by other network participants. The owner is initialized in the *constructor* when the contract is created. The *constructor* gets executed exactly one time at contract creation. The *msg.sender* global variable references an address object, specifically the address of the externally owned account (EOA), who signed and broadcasted the contract creation transaction. Inside the *constructor* code, the transaction sender's address is assigned to the *contractOwner* state variable.

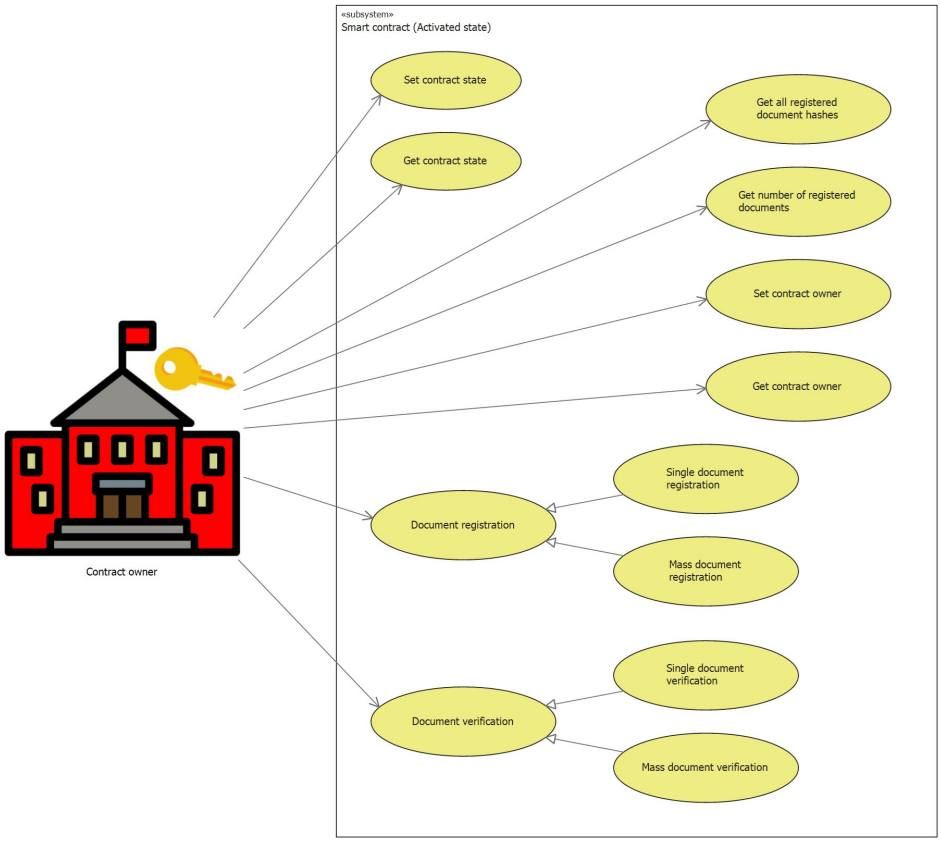


**Figure 1.** Use case diagram with the contract owner actor and its use cases.

The function called *getOwnerAddress* returns with the address of the current contract owner. It is marked as *external* in the function declaration, therefore it can only be called externally, outside of the contract. Using *public* visibility is

also a viable option, although *public* functions cost more gas to call than *external* functions. The function is also a view type function, which means that it will not modify or add new data to the contract storage memory, only reading from it. There might be a need for changing the owner, that is why the contract has a function called *setOwnerAddress* which takes care of this task. The caller has to pass an address type variable as an argument to the function. The *newContractOwner* parameter's value is then assigned to the *contractOwner* state variable, changing the owner of the contract. The *setOwnerAddressEvent* event helps contract activity monitoring by emitting the correct event when the function's code is executed [14]. This particular event contains the transaction sender, the new contract owner's address, a timestamp in epoch seconds, and block height data [7].

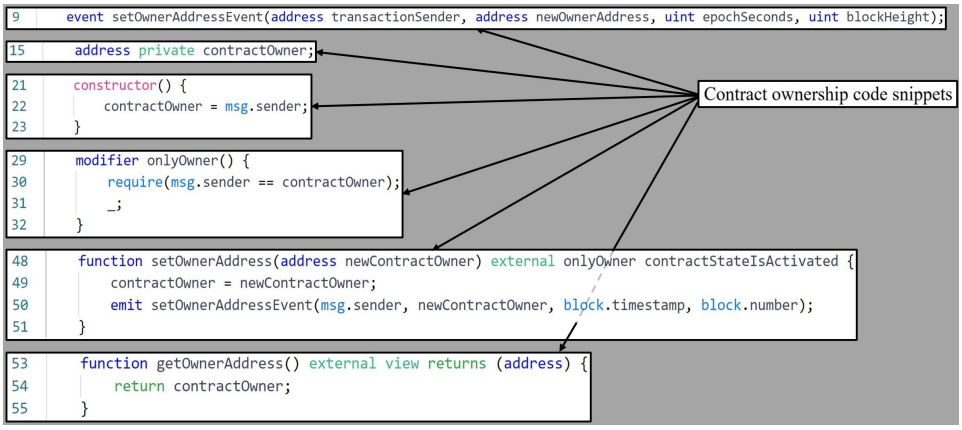


**Figure 2.** Smart contract ownership Solidity code snippets.

We also need to validate that the owner called the given function, for example in the case of document registration or when transferring ownership to another EOA address. This is where the modifier called *onlyOwner* comes into play, which can ensure that only the owner can successfully call a function declared with the *onlyOwner* modifier. The *require* statement contains a statement that the transaction signer is the same as the contract's current owner stored in the *contractOwner* state variable. If that statement is true, then the _; syntax will absorb the code of the called function and get executed. On the other hand, if the *require* statement returns a false logical value, then the function which has the *onlyOwner* modifier inside its declaration will not execute and the transaction will get reverted. This mechanism ensures that the ownership permission is always enforced whenever the method declaration contains the modifier, also it makes it efficient for us to create and manage permission levels safely with total control.

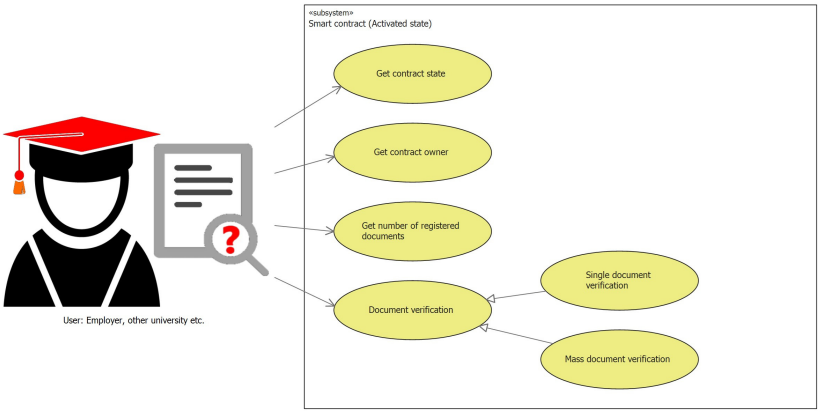


**Figure 3.** Use case diagram with regular user actor and its use cases.

## 2.6. Contract state switch mechanism

The smart contract provides an on-off switch mechanism regarding the state of the contract's specific functions. Another way to turn a contract off is implementing the special *selfdestruct* method, although this has many drawbacks. Firstly, after the self-destruction function is called and takes place, the contract's code and storage memory will be erased permanently, with no way of reverting it. Secondly, it would be counter-productive to use self-destruction, due to the fact that our use case is based on data immutability, which would be somewhat damaged, should we implement the *selfdestruct* function [12]. Deactivating the contract means that some functions will stop working and will revert when called. Such functions are declared with the modifier called *contractStateIsActivated*. The contract can be easily turned on by the owner again. Deactivating the smart contract's state will not affect the code and storage memory of it, document hashes are resistant to any kind of manipulation or tampering.

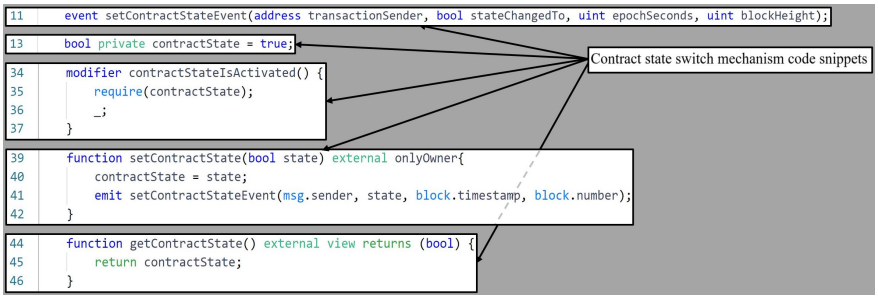


**Figure 4.** Smart contract state switch mechanism Solidity code snippets.

The above-explained mechanism is made possible by a state variable, a modifier, and several functions. The bool primitive data type variable called *contractState* stores the logical value representing the smart contract's state. The initialization is not done in the *constructor*, instead, it takes place right after the declaration, this way it consumes less gas to create the contract. Declaring the visibility to *private* is optional, the contract's state can still be monitored using any block explorer by any participant. *Private* variables are still visible to all network participants, although such variables cannot be used anywhere outside the contract. The next crucial component of the on-off switch is the modifier called *contractStateIsActivated*. The *require* statement in the modifier's body ensures that if the contract's state is deactivated, the called function will be reverted, if the given function's declaration statement contains the *contractStateIsActivated* modifier. If the *contractState* variable is set to true logical value, then the require statement returns true, meaning, that the _; syntax will absorb the code of the called function, executing it. On the other hand, in case the contractState variable is false, then the call will be reverted.

The *setContractState* function is used to set the contract's state by passing a bool logical value when calling it. We must ensure that only the owner of the contract has the right to change the state of the contract, that is why the function declaration contains the *onlyOwner* modifier. When the owner calls the *setContractState* function, a bool argument must be passed along with the call. The parameter is then assigned to the state variable that stores the contract's state, namely the *contractState* variable. At last, an event is emitted in case of successful execution, called *setContractStateEvent*. This particular event will emit data such as the transaction sender's address, the bool value the state has been changed to, in addition to the time in epoch seconds in which the transaction took place, lastly the block height data. The function called *getContractState* is included in the code to make it easy for users to read out the contract state value without having to check through events and the transaction history of the contract. This is an optional function, as the state variable could be declared with *public* visibility, in which case the get function is generated by default.

## 2.7. Document hash registration onto the blockchain

The main objective of the smart contract is to register documents, such as diplomas and certificates onto the blockchain. Precisely we use the smart contract to store document hashes on the blockchain, which is the contract's storage memory. There are two data structures, a map with key-value pairs called *documentMapping*, and a byte32 dynamic array with the name of *documentHashes*. The most important data structure is the mapping, which is the heart of the smart contract. An important note about the mapping structure is that it is very gas efficient. Every registered document hash acts as a key in the mapping, which has a bool value assigned to it acting as a value. When a document hash is registered, it means that the hash key will have a true logical value associated with it. A document hash that has not been already registered will have a false value in the key-value pair in the mapping data

structure. The dynamic array on the other hand stores all the registered hashes for convenience reasons mostly. It would be still possible to determine all the document hashes registered in the smart contract by monitoring the transaction history and event with a simple block explorer or full node. Another important fact is that we cannot retrieve a hash from the mapping structure, as we can only ask for the value with the correct key. In the case of document authenticity verification, we can pass the document hash as an argument to the correct function in order to determine if the bool value associated with it is true or false, registered or not. In this current implementation, there is no way to change the key-value pair's value element from true to false. Such a function could be easily implemented, although it would somewhat damage the document's immutability property. A bytes32 value represents a document hash, 32 bytes are 256 bits. The SHA-256 hash algorithm is perfectly suited to generate 256-bit hash values safely from document data concatenations.


```
7   event documentRegistrationEvent(address transactionSender, bytes32 documentHash, uint epochSeconds, uint blockHeight);
8   event massDocumentRegistrationEvent(address transactionSender, bytes32[] arrayOfDocumentHashes, uint epochSeconds, uint blockHeight);
17  mapping(bytes32 => bool) private documentMapping;
19  bytes32[] private documentHashes;
57  function getNumberOfRegisteredDocuments() external view returns (uint) {
58      return documentHashes.length;
59  }
61  function getAllRegisteredDocumentHashes() external onlyOwner contractStateIsActivated view returns (bytes32[] memory) {
62      bytes32[] memory allRegisteredDocumentHashes = new bytes32[](documentHashes.length);
63      for(uint i = 0; i < documentHashes.length; i++) {
64          allRegisteredDocumentHashes[i] = documentHashes[i];
65      }
66      return allRegisteredDocumentHashes;
67  }
82  function registerDocument(bytes32 documentHash) external onlyOwner contractStateIsActivated {
83      if(documentMapping[documentHash] == false) {
84          documentHashes.push(documentHash);
85      }
86      documentMapping[documentHash] = true;
87      emit documentRegistrationEvent(msg.sender, documentHash, block.timestamp, block.number);
88  }
90  function massRegisterDocuments(bytes32[] memory arrayOfDocumentHashes) external onlyOwner contractStateIsActivated {
91      for(uint i = 0; i < arrayOfDocumentHashes.length; i++) {
92          if(documentMapping[arrayOfDocumentHashes[i]] == false) {
93              documentHashes.push(arrayOfDocumentHashes[i]);
94          }
95          documentMapping[arrayOfDocumentHashes[i]] = true;
96      }
97      emit massDocumentRegistrationEvent(msg.sender, arrayOfDocumentHashes, block.timestamp, block.number);
98  }
99  }
```

**Figure 5.** Smart contract document hash registration Solidity code snippets.

In order to register a single document hash, the owner of the contract must call the function named *registerDocument*. The function takes one argument with a data type of bytes32, which represents the 256-bit document hash. The contract's state must be set to true, as we can see in the function declaration that it contains the *contractStateIsActivated* modifier, in addition to the *onlyOwner* modifier, which ensures that only the owner could register a document with the smart contract. Next, the function starts with checking if the hash has been already registered or not. We must find out the value element of the key-value pair with the hash

parameter. If the hash has the logical value true associated with it, that means that the hash has been already registered by the institution. If it is false, then we proceed to add it to the *documentHashes* dynamic array with the *push* method. This way we ensure that there will not be duplicate hash values in the array. Next, the function performs the most important core statement of the smart contract, which is document registration. This is done by changing the value element of the key-value pair of the argument which has been passed to the function in the mapping data structure. Setting the value to true means that the document hash has been registered and that the document will be stored on the blockchain forever, never to be removed or modified in any way. There is also no way to roll back registrations, setting the values of the key-value pairs to false. Finally, the function emits an event named *documentRegistrationEvent*, which contains the transaction sender's address, the document hash which has been registered, in addition to the epoch seconds, and block height data for optimal traceability.

There is a function implemented to be able to mass register documents. This is particularly important for universities and other institutions who need to register documents such as diplomas in big batches, periodically. The function named *massRegisterDocuments* takes a dynamic, bytes32 data type array as an argument, which contains the document hashes. The function must be called by the contract owner in order to execute successfully. Also, the contract must be activated too. The function immediately starts a cycle in order to iterate through all the bytes32 data elements of the passed dynamic array. In each iteration, we must check if the given hash is already registered or not. If the key-value pair has a value element of false associated with the hash key, then the hash is pushed into the *documentHashes* state object variable in the smart contract storage memory. In each iteration, we must also register the document hash the same way we do in the previous function, by setting the bool value from false to true corresponding to the hash as a key in the key-value pair. After the cycle is finished, an event named *massDocumentRegistrationEvent* is emitted with the following data elements: The transaction sender's address, the dynamic array argument contents, block timestamp data, and block height data. There are two additional functions included. First, the function named *getNumberOfRegisteredDocuments*. As the name suggests, it will return with the total number of stored hashes in the contract storage, precisely the length property of the *documentHashes* state object variable. It is ensured that the dynamic array will not contain any duplicate bytes32 values. Lastly, we have the function named *getAllRegisteredDocumentHashes*. When called by the owner, it will return the elements of the *documentHashes* array. The *memory* keyword must be used in case of arguments, local variables, and return values in methods. Memory variables are created at runtime and exist only while the method is being executed, after that the memory variables are released. A popular analogy is comparing the two memory types to the computer random access memory and hard drive. Memory data is only stored temporarily like RAM, and is volatile, meaning that after function execution the memory variables are released. On the other hand, *storage* variables are non-volatile and can be always retrieved by reading the

contract's storage area. From a gas cost perspective, reading from the contract's storage is expensive compared to using memory variables that are only temporarily allocated. Another interesting aspect of this is when we reference storage variables inside a function. In Solidity, that action is called a *local lookup* operation, and it does not create new storage, it is just a reference to a storage variable that has been already allocated in the contract's storage area. It is impossible to create new storage variables inside a function. When a function takes a memory variable argument and assigns the parameter to a referenced storage state variable, there is no need for new storage allocation, as it is done already at the contract construction level. The question of using memory or storage variables comes down to whether we need to store a variable on-chain or is only needed at runtime, and also gas cost considerations are not to be underestimated.

## 2.8. Document authenticity verification

One of the major use cases of the smart contract is to verify documents based on the unique document fingerprint hash. Document verification is available to all users, it does not require special permissions or even an activated smart contract state. The functions responsible for this process are free to call, as they are read-only functions. Read-only methods only access the storage memory area of the smart contract to retrieve data, but do not modify or add any new data to it, therefore executing these functions does not cost anything as the state of the blockchain and the smart contract stays the same.

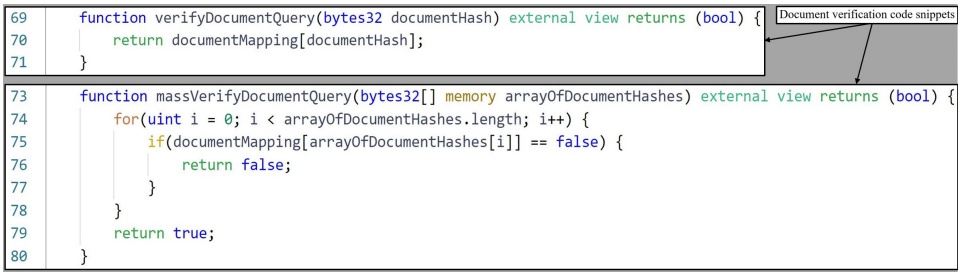


**Figure 6.** Smart contract document authenticity verification Solidity code snippets.

The function *verifyDocumentQuery* has the ability to verify one document. The caller must pass a bytes32 document hash argument to the function. The method then returns with the value element of the key-value pair, where the *documentHash* parameter is the key. If the passed hash has been already registered by the owner of the contract, then the value element of the key-value pair will be true, otherwise, it will return with false. There is also a way to verify multiple documents with one function call. This is done by calling the *massVerifyDocumentQuery* function and passing a valid bytes32 dynamic array of document hashes. Next, the function starts to iterate through the list of hashes, checking each of them if they have been

registered or not. If one of the document hashes has not been registered in the contract, or in other words the key-value pair holds a value of false associated with the given hash key, then the function returns immediately with false. If all of the document hashes are valid and registered, then the function returns with true.

## 2.9. Fallback method

```
10    event fallbackEvent(address transactionSender, string fallbackMessage, uint epochSeconds, uint blockHeight);
25    fallback() external {
26        emit fallbackEvent(msg.sender, "fallback method activated: Wrong function prototype or empty ether call", block.timestamp, block.number);
27    }
```

Fallback function code snippets

**Figure 7.** Smart contract fallback function Solidity code snippets.

The *fallback* function is a nameless function, which has no arguments, does not return a value. The reason why we need a *fallback* function is to prevent the contract from receiving ether from users by mistake. If we declare the *fallback* function without the *payable* keyword in the declaration, then the function will throw an exception in case the contract receives ether without any function calls. It is also called when the transaction contains a function call and the function identifier is constructed incorrectly, calling a non-existent function instead [19]. In these cases, the *fallback* function is called, emitting the event named *fallbackEvent*, which has a transaction sender address, fallback message, block timestamp, and block height data. Another constraint is that the *fallback* function must be declared with *external* visibility. If we would mark the *fallback* function with the *payable* keyword, then the contract would be able to receive ether as payment, although in our use case this is unnecessary. Only one *fallback* function can be defined in a contract. The main use of the function in our implementation is to avoid loss of funds for users who send plain ether to the contract, such transactions will be reverted. Another use of this method is to receive donations or payments without having to supply a function with the transaction.

# 3. Conclusion

We aimed to develop a smart contract, which enables decentralized document registration and verification for the education sector and academia on the Ethereum network, without a third party. Our main objective was to come up with a powerful solution of reducing document forgery cases. We successfully deployed the smart contract on the Ropsten Ethereum test network. All the implemented logic, mechanisms, and functions performed as intended, according to the previously set expected results. Another goal was to keep gas costs low for all use cases of the system. After proper empirical evidence, we can safely say that there were no unexpected gas spikes, and the gas consumption of the functions is optimal thanks to the simplistic, yet efficient and safe design. There is still much room for improvement regarding the capabilities of the smart contract.

One of these is implementing a multi-signature scheme for the system. With a secure and gas-efficient multi-signature mechanism in place, we could improve the security of the contract even further, by having multiple owners and therefore required signatures for successful function execution such as document registration or contract deactivation, or even ownership transfers. Another requirement might be a feature that enables the contract owner to call back documents, which have been proved to contain errors, or in rare cases, the document owner has become ineligible for ownership of the specific document. The next step of evolution for our smart contract is the deployment on other blockchains. A great potential blockchain candidate is called Avalanche [8] because it offers much greater transaction block inclusion and finality speed, lower transaction fees. Instead of using the Proof-of-Work consensus algorithm, Avalanche implemented a Proof-of-Work mechanism instead, which has several benefits, such as greater decentralization, neutrality, and stronger scalability, in addition to being more resistant to 51% attacks [4]. Although Ethereum has the most robust network and node infrastructure, in addition to having the longest track record, and is still the dominant blockchain by adoption, volume, users, developers, and decentralized applications.

The on-chain part of the system is the smart contract. There is still a need for several off-chain components, such as a database server, a web server, and unique, in-house developed programs. A database server is required to store all the registered documents along with the hashes for positive redundancy and traceability. Remember, that the smart contract does not store the document data directly, only the calculated hash of the document to prevent data leakage and maintain privacy. Another off-chain component is a web server, to host a web interface for users to interact with the contract more easily. Without a user-friendly web interface, the users would only have the choice to interact with the smart contract manually with a proper wallet and block explorer. Having more than one way to reach and communicate with the contract's functions is only beneficial and increases the security of the system. Another off-chain component that would be ideal, is a program that calculates the document hashes and checks for collisions and other potential errors. The software would also prepare the arguments to be passed during a function call. Thanks to the invention of blockchain and smart contract technology, now we have the tools necessary to fight against document forgery in an unprecedented way.

# References

[1] A. Alammary, S. Alhazmi, M. Almasri, S. Gilani: *Blockchain-Based Applications in Education: A Systematic Review*, Applied Sciences 9 (June 2019), p. 2400, doi: https://doi.org/10.3390/app9122400.

[2] P. Bhardwaj, Y. Chandra, D. Sagar: *Ethereum Data Analytics: Exploring the Ethereum Blockchain*, Sept. 2021.

[3] V. Buterin: *A Philosophy of Blockchain Validation*, 2020, url: https://vitalik.ca/general/2020/08/17/philosophy.html.

[4] V. Buterin: *Why Proof of Stake*, 2020, url: https://vitalik.ca/general/2020/11/06/pos2020.html.

[5] G. Canfora, A. D. Sorbo, Sonia, A. V. Laudanna, C. A. Visaggio: *Profiling Gas Leaks in Solidity Smart Contracts*, Aug. 2020.

[6] *Chainlink: Blockchains and Oracles: Similarities, Differences, and Synergies.* 2021, URL: https://blog.chain.link/blockchains-oracles-similarities-differences-synergies/.

[7] *Chainlink: Events and Logging in Solidity*, 2021, URL: https://blog.chain.link/events-and-logging-in-solidity/.

[8] *Chainlink: How to Build and Deploy an Avalanche Smart Contract.* 2021, URL: https://blog.chain.link/how-to-build-and-deploy-an-avalanche-smart-contract/.

[9] *Chainlink: What Is a Smart Contract?*, 2021, URL: https://chain.link/education/smart-contracts.

[10] *Chainlink: What Is Blockchain Technology?*, 2020.

[11] G. Chen, B. Xu, M. Lu, N.-S. Chen: *Exploring blockchain technology and its potential applications for education*, Smart Learning Environments 5 (Jan. 2018), DOI: https://doi.org/10.1186/s40561-017-0050-x.

[12] J. Chen, X. Xia, D. Lo, J. Grundy: *Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum*, May 2020.

[13] H. Farid: *An Overview of Perceptual Hashing*, Journal of Online Trust and Safety 1.1 (2021), DOI: https://doi.org/10.54501/jots.v1i1.24.

[14] Á. Hajdu, D. Jovanović, G. Ciocarlie: *Formal Specification and Verification of Solidity Contracts with Events*, May 2020.

[15] N. Kannengiesser, S. Lins, C. Sander, K. Winter, H. Frey, A. Sunyaev: *Challenges and Common Solutions in Smart Contract Development*, Oct. 2021, DOI: https://doi.org/10.1109/TSE.2021.3116808.

[16] W. Macharia: *Cryptographic Hash Functions* (May 2021).

[17] G. A. Pierro, H. Rocha: *The Influence Factors on Ethereum Transaction Fees*, in: May 2019, pp. 24–31, DOI: https://doi.org/10.1109/WETSEB.2019.00010.

[18] B. Preneel: *Analysis and Design of Cryptographic Hash Functions* (2013), pp. 1–30.

[19] S. Rezaei, E. Khamespanah, M. Sirjani, A. Sedaghatbaf, S. Mohammadi: *Developing Safe Smart Contracts*, in: July 2020, pp. 1027–1035, DOI: https://doi.org/10.1109/COMPSAC48688.2020.0-137.

[20] J. Udvaros, N. Forman, S. M. Avornicului: *Agile Storyboard and Software Development Leveraging Smart Contract Technology in Order to Increase Stakeholder Confidence*, Electronics 12.2 (2023), DOI: https://doi.org/10.3390/electronics12020426.

[21] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, F.-Y. Wang: *Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends.* IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS: SYSTEMS 49.11 (2019), DOI: https://doi.org/10.1109/TSMC.2019.2895123.

[22] Z. Zheng, S. Xie, H.-N. Dai, Weili, X. C. Chen, J. Weng, M. Imran: *An Overview on Smart Contracts: Challenges, Advances and Platforms*, Dec. 2019.