2018 IEEE Confs on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, Congress on Cybermatics

SmartEdge: A Smart Contract for Edge Computing

Kwame-Lante Wright, Martin Martinez, Uday Chadha, Bhaskar Krishnamachari
Viterbi School of Engineering
University of Southern California
Los Angeles, CA
{*kwamelaw, mart698, uchadha, bkrishna*}@*usc.edu*

***Abstract*—**
Edge computing has emerged as an effective offloading strategy for constrained devices. It enables low-capability devices to leverage nearby resources for assistance with computationally-intensive tasks. We envision a future where Internet of Things (IoT) devices may autonomously transact with other more powerful devices to request such offloading services. We believe blockchain-based technologies can help facilitate this process by tracking usage and managing payments. In this work we introduce SmartEdge, an Ethereum-based smart contract for edge computing and show that it is a low-cost, low-overhead tool for compute-resource management.

***Index Terms*—IoT, edge computing, smart contracts, blockchain**

1. Introduction

Cloud computing has emerged as a popular platform to facilitate the storage and processing of information from Internet of Things (IoT) devices [1]. In recent years an alternative approach has been gaining momentum, namely edge computing [2], which is a set of techniques for processing information either locally or in close proximity to the devices that generate the information, as opposed to a centralized data center.

Recent work on offloading has discussed how edge computing can manage limited resources and provide mechanisms for mobile devices to leverage them effectively [3]. The telecommunications study presented by Chen and Xu [4] explores how the usage of small cell base stations (SBSs) as edge computing enablers can improve latency and the location awareness of resource utilization. However, due to the expected higher demand in the future, there is a need to relieve the high computational workload that may be required for particular SBSs. To do so, Chen and Xu [4] talk about the opportunity for creating coalitions of these devices to share their resources, which also includes a payment-based mechanism for proportionally fair utility as well as a trust network for minimizing security risks. In contrast, our work on SmartEdge aims to provide an easier way to handle resource allocation without preexisting relationships through the utilization of smart contracts.

There are a few prior works that address blockchain and edge computing. Xiong et al.'s work [5] is about the creation of a mobile chain using edge computing as a complement of previous applications in the areas of healthcare, finance, and so on. This study analyzes a prototype model in two different scenarios: one where the relation between mining rewards and optimal edge service price is fixed, and another where this is variable and provides results which can be leveraged by these types of providers when defining an optimal resource management policy.

Stanciu's study [6] explores the implementation of a blockchain-based technology to support distributed control systems in the realm of edge computing, using Hyperledger Fabric to generate function blocks as smart contracts which will be executed by the blockchain and delegate the tasks and resources using Docker containers and Kubernetes for the orchestration of container execution.

As we can see, previous efforts have discussed edge computing as an opportunity to solve the computational resource limitation existent in IoT systems. Additionally, these studies have presented blockchain as a viable platform to develop a mechanism that can resolve this issue as well as provide additional features to the system. However, to our knowledge, this is the first work to present the design and implementation of a smart-contract for edge computing based on the public permissionless blockchain Ethereum.

2. Smart Contracts on Ethereum

The phrase smart contract was first coined by Nick Szabo [7]. It refers to the verification, monitoring and execution of contracts including transfer of money using a software implementation. While it is possible to implement certain simple smart contracts in the original Bitcoin protocol [8], Ethereum was the first open Blockchain to be explicitly designed with programmability of smart contracts in mind [9]. It is designed to be a state transition machine with code being executed simultaneously by all miners on the decentralized Ethereum Virtual machine [10]. Ethereum allows in principle for Turing-complete computations.

We give a brief overview on how smart contracts work on Ethereum. A smart contract on Ethereum can be coded using a specialized programming language called Solidity [11] (there are alternative approaches too, but Solidity is by far the most popular today), and deployed through an initial transaction. Smart contracts have their own specific address. A contract consists of state variables and functions. The functions can be called by additional transactions addressed to the contract to trigger changes in the state variables, make payments, etc. In order to support the storage, computation and communication costs of executing smart contracts there are fixed "gas" fees associated with each operation that must be paid by the caller.

For any transaction there is a gas limit, which should be more than the amount of gas needed for the transaction (if the gas limit is less than the needed amount, an out of gas exception occurs and the transaction is not executed). Further each transaction must also have a gas price (measured in GWei per gas) associated with it, which is chosen based on current market conditions to be high enough to incentivize the quick inclusion of the transaction into the blockchain. Roughly speaking, when there is network congestion, the gas price needed for quick inclusion of the transaction is higher.

Smart contracts have certain key limitations that must be kept in mind. One is that they are only triggered to change state by transactions sent to the contract address, and don't have any other automatic way to connect to events outside the chain. Second, they also cannot communicate directly to a server or piece of code that is outside the blockchain. However, they can emit events which can be subscribed to from outside code using the Ethereum API. And ultimately because of the gas costs and gas limits per block (not a fixed limit, but based on a moving average of gas usage over a window), they cannot be arbitrarily complicated.

3. Design and Implementation of the SmartEdge Smart Contract

We break down the design of a smart contract into three key steps and describe them below in the context of our design of SmartEdge[1]:

1) identify the parties involved in the smart contract
2) identify key states in the lifetime of the smart contract
3) identify and define the methods that trigger state transitions

3.1. Parties Involved

To design a smart contract it is helpful to first identify the parties involved in transactions. The design of the contract for SmartEdge assumes that there are two types of parties who interact through Ethereum, as shown in Figure 1. The first party is the compute node. The compute node refers to the computing resource that will be made available through the SmartEdge contract to perform work. The second party is the data node, which we also refer to as the counterparty. The data node possesses data that needs processing and the program that should be used to process it. In this paper, we collectively refer to the data and the program as a job.

1. The SmartEdge source code will be made available online at https://github.com/ANRGUSC/SmartEdge

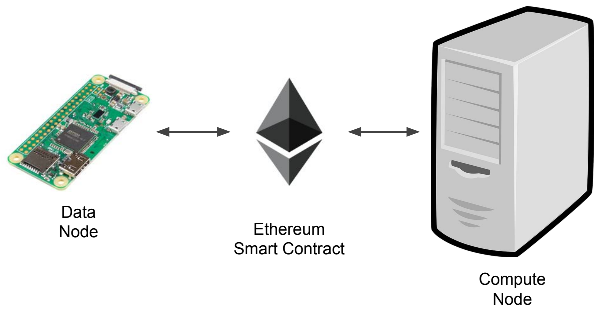

Figure 1. Example of SmartEdge participants

3.2. State Machine

Second, it is helpful to identify the key states that occur in the lifetime of the smart contract. During the lifetime of a job that is processed through SmartEdge, the contract will proceed through a series of states. These states are shown in Figure 2. Each state requires that either the compute node or data node take an action to move the contract into the next state.

There are a total of five states in the SmartEdge contract. We describe them here:

- **Unavailable**: In this state, the compute node is not available to take any job requests.
- **Available**: The compute node is announcing a price at which it is prepared to allocate its resources. It will accept the first job that comes along.
- **Pending**: The compute node has received a job request and is considering whether or not to accept it. The compute node must first retrieve the data and program from the specified location. The location, represented by a URL, could refer to a folder located on the data node itself or some other accessible resource with the files. We chose to use a URL because anything passed to the contract will be stored on the blockchain. It is impractical to store files directly on the blockchain so we use the URL here instead as an indirect reference. While considering whether or not to accept the job, the compute node could also take into consideration the reputation of the data node as there may be malicious users. Reputation is not monitored by the SmartEdge contract itself but we imagine a separate system could be utilized for this purpose.
- **Computing**: In this state, the compute node is busy processing a job.
- **Completed**: At this phase, the compute node has successfully completed a job and is waiting for confirmation from the data node to conclude the interaction. The compute node will provide a URL

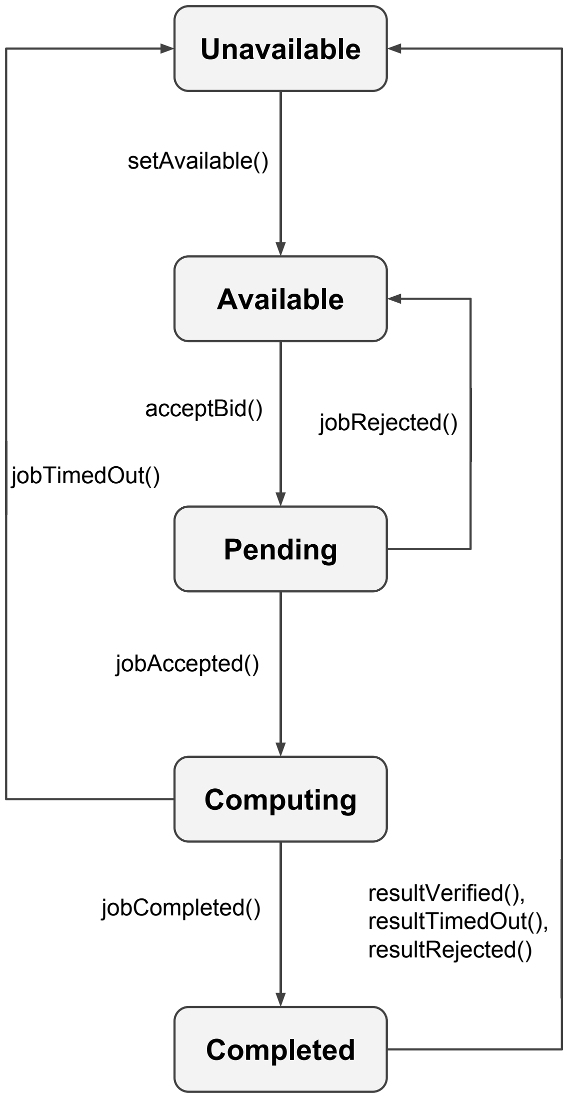

Figure 2. SmartEdge State Machine

for the data node to retrieve the result. The data node has the option of of accepting or rejecting the result which we will discuss in more detail later.

3.3. Contract Methods and Events

Third, it is important to identify the methods (which are called by corresponding transactions sent to the contract) that trigger changes between the states. In this section we describe the methods of our SmartEdge contract. Calling these methods trigger the state transitions as shown in Figure 2 and will also fire off Events that can be used by applications outside of the blockchain. We will discuss in Section 4 how we use these Events in our experiments.

When a new SmartEdge contract is first instantiated, it starts off in the `Unavailable` state. Through the *setAvailable()* method, the compute node is able to set the bid price for computation. The bid price represents a fixed cost per unit of time. This price is set under the assumption that a known and fixed resource will be allocated for processing for the duration of a job handled by the contract. Since it is not necessarily known ahead of time how long a job may take, we believe this pricing scheme provides a fair valuation of the work performed. The *setAvailable()* method also verifies that the compute node has placed enough funds in escrow. The compute node's escrow is used to assess any penalties for falsified work. The amount required for escrow is a parameter that can be specified prior to contract deployment. After all conditions for availability are satisfied, the *setAvailable()* method will change the contract state to `Available` and emit the MadeAvailable event.

While in the `Available` state, a counterparty may begin a request for work to be performed through the *acceptBid()* method. When calling this method, the counterparty will send a URL where files pertaining to the job can be found. This includes the data and the processing program. The *acceptBid()* method also takes a payment from the counterparty to hold in escrow as the computation is being performed. The cost of the job will later be deducted from this amount. The minimum amount required for the counterparty escrow is established prior to contract deployment. If the escrow requirements are satisfied, the contract will then enter the `Pending` state.

In the `Pending` state, the contract owner has the option of accepting and rejected a proposed job. We include this capability to allow the owner to assess the complexity of the job and the reputation of the counterparty. If the owner does not want to accept the job, the *jobRejected()* method may be called taking the contract back into the `Available` state. The counterparty's escrow will be returned in this case. If the owner accepts the job, then the *jobAccepted()* method will be called. This method will establish the start time of the job and send the contract into the `Computing` state.

The compute node is expected to be processing the job while in the `Computing` state. If for some reason the compute node never finishes the job or is simply taking too long, then the job can be cancelled by the counterparty using the *jobTimedOut()* method. This method will refund the counterparty's escrow and place the contract back into the `Unavailable` state. The amount of time required to elapse before a timeout can occur is established in the contract prior to deployment. If the job is successfully completed, then the compute node will call the *jobCompleted()* method. This establishes the job's finish time, announces the location of the result, and places the contract into the `Completed` state.

When in the `Completed` state, one of three things may happen. Firstly, in an ideal scenario, the counterparty will check the results and be satisfied. By calling the *resultVerified()* method, the counterparty pays the costs of the job with their escrow and is refunded the remaining balance. The contract is then taken back into the `Unavailable` state. Secondly, if the counterparty believes the results are fraudulent, the *resultRejected()* method may be called. This will cause all of the counterparty's escrow to be refunded and the contract will go into the `Unavailable` state. Lastly, if the counterparty takes too long to respond regarding the result, then the compute node may call the *resultTimedOut()* method. The compute node will receive payment for the work that was completed and the remaining balance of the counterparty's escrow will be refunded. The amount of time required for a result timeout is configured in the contract prior to deployment. As with the other two related actions, the contract will then enter the `Unavailable` state.

4. Experimental Setup

Our testbed consists of two components as shown in Table 1. We use a Raspberry Pi 3 to serve as our data node and a more powerful desktop PC as our compute node. For development purposes, our compute node also serves as our Ethereum node. We create a test blockchain using Ethereum's Truffle development framework [12]. As shown in Figure 3, the two devices are connected wirelessly through a WiFi router.

Device	CPU	RAM
Data Node (Raspberry Pi 3)	ARM Cortex-A53 (1.2 GHz)	1 GB
Compute Node (Desktop PC)	Intel Core i7-4770 (3.4 GHz)	16 GB

TABLE 1. HARDWARE SPECIFICATIONS

To automate the actions taken by the compute node and data node during our experiments, we utilize contract events. When emitted, these events are stored in a transaction's log and can be used to trigger callbacks in interested applications. We implement event callbacks in our testing framework using the Web3.js library [13]. Examples of the events used in our contract are shown in Listing 1. Events are the preferred method for monitoring a smart contract because they provide push-based notifications as opposed to pull-based notification where an application would constantly have to poll the blockchain to check for changes.

```
// Transition Events
event MadeAvailable();
event BidAccepted(string url);
event JobRejected();
event JobAccepted();
event JobCompleted(string url);
event JobTimedOut();
event ResultVerified();
event ResultTimedOut();
event ResultRejected();
```

Listing 1. Contract events

5. Results

Through a series of experiments we seek to evaluate the effectiveness of our proposed smart contract and demonstrate that it can support edge computing in an IoT environment. In particular, we aim to assess if SmartEdge achieves the following properties:

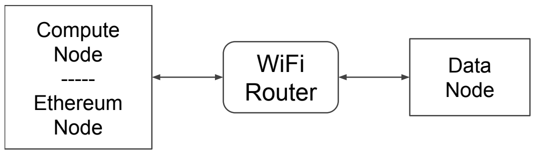

Figure 3. Network topology

- **Low-overhead**: Given that the purpose of edge computing is to speed up processing, SmartEdge should not contribute significantly to the delay of executing a job.
- **Low-cost**: As a smart contract, there are transaction costs associated with using SmartEdge. These costs should not be significant relative to the value it provides.

5.1. Transaction Costs

Table 2 shows the gas costs associated with using the SmartEdge contract. For the compute node owner, the SmartEdge contract will cost about $3.96 USD to deploy on the Ethereum blockchain. We believe this to be a reasonable upfront cost for the contract. The contract is reusable so the owner should eventually be able to recover the deployment cost. The remaining contract methods are all an order of magnitude cheaper, costing $0.18 USD or less per call.

Method	Gas	Ether	USD
constructor	1,832,908	0.005498724	$3.96
setAvailable()	83,997	0.000251991	$0.18
acceptBid()	74,942	0.000224826	$0.16
jobRejected()	31,161	0.000093483	$0.07
jobAccepted()	47,848	0.000143544	$0.10
jobCompleted()	69,270	0.000207810	$0.15
jobTimedOut()	23,116	0.000069348	$0.05
resultVerified()	30,030	0.000090090	$0.06
resultTimedOut()	27,422	0.000082266	$0.06
resultRejected()	29,578	0.000088734	$0.06

TABLE 2. TRANSACTION COSTS (AS OF MAY 3, 2018)

5.2. Overhead

To evaluate the overhead of SmartEdge, we first created workload to test with. We developed a job that involves reading integers from a file line-by-line and determining their multiplicative factors. The input file contains numbers that are randomly selected from the range 1,000 - 2,000,000,000. The small version of this file (input.txt) consists of 10,000 integers while the large version consists of 100,000. A python script (factor.py) determines the factors using the `pyprimes` library. The file sizes for all files are shown in Table 3. The result files corresponding to each input file are shown as well.

We measured the execution time of the factorization job using the data node, compute node, and SmartEdge. The results are shown in Table 4. The job takes over 3 minutes

Transferred Files	Size
input.txt (small)	102 KB
input.txt (large)	1,020 KB
factor.py	494 B
result.txt (small)	198 KB
result.txt (large)	1 MB

TABLE 3. File Sizes

to execute on a Raspberry Pi 3. However, when the job is executed using SmartEdge, it only takes 8.6 seconds. There is an overhead of 2 seconds compared to executing the job directly on the compute node. It should be noted that this 2 second overhead includes the time it takes to transfer the job to the compute node and the result back to the data node.

Platform	Execution Time (seconds)
Data Node	182.9
Compute Node	6.6
SmartEdge	8.6

TABLE 4. Factorization Task Execution Time (small input)

Figure 4 shows the percent of time the SmartEdge contract spends in each state. As expected, the `Computing` state dominates the time which is the desired behavior. However, we realized that in the Truffle development blockchain, blocks are mined instantly which is not an accurate representation of mining time in practice. We re-ran the experiment using Truffle's Ganache blockchain, with the mining time set to the Ethereum average of 14 seconds. The results are shown in Figure 5. In this figure we can infer that the mining time for each transaction dominates the time spent in each state of the contract and is an important consideration when choosing to offload a job.

We ran the experiment one more time to see how the durations would be affected if the execution time of the job was higher relative to the block mining time. The execution time of the job with the larger input file is 67 seconds when directly run on the compute node. Indeed, we can see in Figure 6 that the percentage of time spent in the `Computing` increased significantly relative to the other states.

We believe that a larger input file and result file would have a similar impact on the `Available` and `Completed` states, respectively, as they would affect the file transfer times. We aim to investigate this hypothesis in future work.

6. Risks and Security Issues

There are some issues in the current design of our contract that we have identified and discuss below.

Results Verification: Due to the computational constraints on smart contracts, we believe it is difficult in general to verify that a job was properly performed through the smart contract itself. While there are techniques for verifiable computing [14], the expensive costs of circuit generation and homomorphic encryption make it generally impractical for this context. For now, we leave it up to

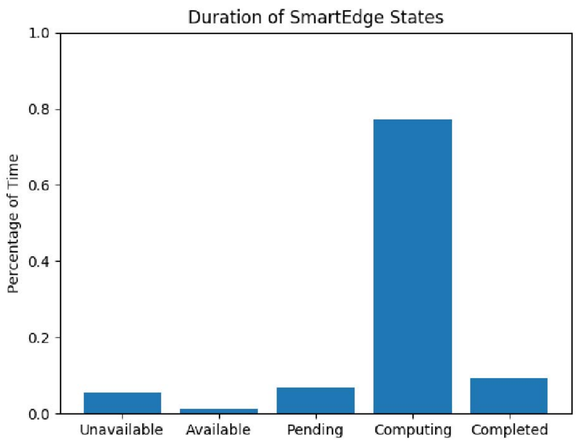

Figure 4. Time spent in each SmartEdge state with instant mining

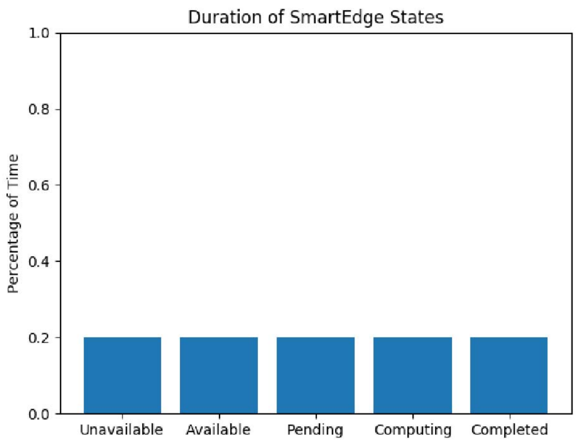

Figure 5. Time spent in each SmartEdge state with 14-second mining

the data node to verify that the result is valid. If it is trivial to check all of the result, then the data node may do so. However, if result verification is also computationally intensive, then the data node can randomly plant some test data in the original input file for which the result is already known. For example, if we assume a similar job structure as in our experiments, then the data node can plant test data at every n^{th} line of the input file. Alternatively, the test data can be placed in a more randomize fashion as long as the data node can keep track of the placement. The data node can then use the result for those particular values as a proxy to determine if the rest of job was actually performed correctly.

On the other hand, the data node may be malicious and so the compute node could be at risk. A malicious data node could reject a result to avoid having to pay for the execution of their job. For this aspect of the problem we believe a

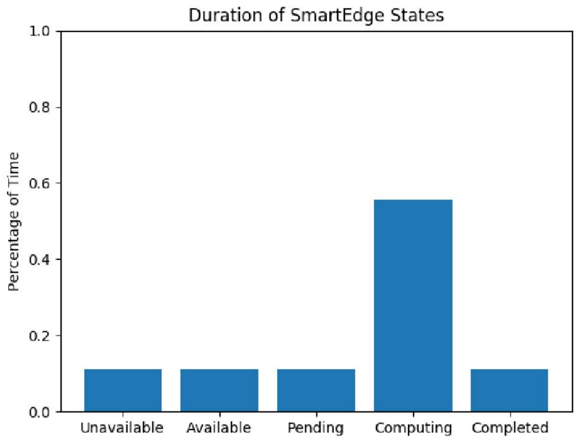

Figure 6. Time spent in each SmartEdge state with 14-second mining and a larger input file

reputation-based system could be useful. A compute node can check the reputation of a data node before accepting their job to avoid such a situation. Another possibility is that the job itself is crafted with some malicious intent. However, we imagine that all jobs processed through SmartEdge will be executed in a sandbox, such as a virtual machine or container, so that it is contained and isolated from the rest of the host system.

Data Exchange: Another security issue in the current design of SmartEdge is the safe transmission of the information between the data node and the compute node. Currently URLs are used to specify the location of files. Once this information enters the smart contract it becomes publically accessible information. This could present a problem if the data being exchanged is sensitive.

7. Future Work

We have several possible improvements we would like to incorporate into our implementation of SmartEdge.

Data Privacy: As mentioned in Section 6, the safe transmission of the information between the data node and the compute node may be a concern. A simple solution to solve this problem would consist of encrypting the URL in such a way that the keys are only known among the participants for each job.

Bid Curves: In the current version of SmartEdge, the bid price for a compute node is fixed. However, we believe a useful modification would be to define the bid price as a function of completion time. The benefits of this change are that, on one hand, it will provide an incentive to the compute nodes for early job completion and, on another hand, it will have as a consequence diminishing returns or penalty for late job completion.

Auctioning Contract: Another possible improvement to the current system is the creation of a separate auctioning contract that can automatically match data nodes with the most appropriate compute nodes. Currently data nodes deal with a compute node directly, but if they want to shop around and consider multiple resources, the auctioning contract can manage that process for the data node automatically.

8. Conclusions

We have presented SmartEdge, a new Ethereum-based smart contract for edge computing. It allows nodes to offload computation in a verifiable manner to edge computing devices belonging to other parties in exchange for payment. We believe SmartEdge will be a valuable tool for IoT applications. While our current implementation serves as a proof-of-concept, we aim to continue development on SmartEdge.

References

[1] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: state-of-the-art and research challenges,” *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[2] M. Satyanarayanan, “The emergence of edge computing,” *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[3] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, “The Case for VM-based Cloudlets in Mobile Computing,” *IEEE Pervasive Computing*, vol. 8, no. 4, 2009.

[4] L. Chen and J. Xu, “Socially trusted collaborative edge computing in ultra dense networks,” *CoRR*, vol. abs/1705.03501, 2017. [Online]. Available: http://arxiv.org/abs/1705.03501

[5] Z. Xiong, Y. Zhang, D. Niyato, P. Wang, and Z. Han, “When mobile blockchain meets edge computing: Challenges and applications,” *arXiv preprint arXiv:1711.05938*, 2017.

[6] A. Stanciu, “Blockchain based distributed control system for edge computing,” *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, pp. 667–671, 2017.

[7] N. Szabo, “The idea of smart contracts,” *Nick Szabos Papers and Concise Tutorials*, vol. 6, 1997.

[8] A. Back and I. Bentov, “Note on fair coin toss via bitcoin,” *arXiv preprint arXiv:1402.3698*, 2014.

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

[10] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.

[11] C. Dannen, *Introducing Ethereum and Solidity*. Springer, 2017.

[12] “Truffle: Ethereum Development Framework,” https://github.com/trufflesuite/truffle, accessed: 2018-05-05.

[13] “Ethereum JavaScript API,” https://github.com/ethereum/web3.js/, accessed: 2018-05-05.

[14] R. Gennaro, C. Gentry, and B. Parno, “Non-interactive verifiable computing: Outsourcing computation to untrusted workers,” in *Annual Cryptology Conference*. Springer, 2010, pp. 465–482.