Scalable and Privacy-preserving Design of On/Off-chain Smart Contracts

Chao Li, Balaji Palanisamy and Runhua Xu

School of Computing and Information, University of Pittsburgh, USA

{*chl205, bpalan, runhua.xu*}*@pitt.edu*

Abstract—The rise of smart contract systems such as Ethereum has resulted in a proliferation of blockchain-based decentralized applications including applications that store and manage a wide range of data. Current smart contracts are designed to be executed solely by miners and are revealed entirely *on-chain*, resulting in reduced scalability and privacy. In this paper, we discuss that scalability and privacy of smart contracts can be enhanced by splitting a given contract into an *off-chain* contract and an *on-chain* contract. Specifically, functions of the contract that involve high-cost computation or sensitive information can be split and included as the *off-chain* contract, that is signed and executed by only the interested participants. The proposed approach allows the participants to reach unanimous agreement *off-chain* when all of them are honest, allowing computing resources of miners to be saved and content of the *off-chain* contract to be hidden from the public. In case of a dispute caused by any dishonest participants, a signed copy of the *off-chain* contract can be revealed so that a verified instance can be created to make miners enforce the true execution result. Thus, honest participants have the ability to redress and penalize any fraudulent or dishonest behavior, which incentivizes all participants to honestly follow the agreed *off-chain* contract. We discuss techniques for splitting a contract into a pair of *on/off-chain* contracts and propose a mechanism to address the challenges of handling dishonest participants in the system. Our implementation and evaluation of the proposed approach using an example smart contract demonstrate the effectiveness of the proposed approach in Ethereum.

I. INTRODUCTION

Creating trust among mutually distrustful participants without involving a trusted third party has been a challenge for several decades. The growth of Bitcoin [11] and the emerging follow-up cryptocurrencies have positioned blockchain as a promising solution for creating trust in a decentralized environment. Smart contracts expand the use of blockchains by allowing mutually distrustful participants to reach an agreement upon the execution results of complex contracts without a trusted third party. In leading smart contract systems such as Ethereum [19] and NEO [12], every single step of the decentralized computation is performed and verified by miners in the blockchain network, who are incentivized by the cryptocurrency rewards to behave honestly. This combination of decentralized computation and cryptocurrency-based incentives has led to the development of a large number of decentralized applications including applications that store and manage a wide range of data [15]. Ethereum [19], the first blockchain system that supports Turing-complete smart contracts, had a peak market cap of $134 billion [3] in 2018.

The Smart Contracts market is estimated to grow at a CAGR of 32% during the period 2017 to 2023 [14].

Current smart contracts are designed to be executed solely by miners and are revealed entirely *on-chain*, resulting in reduced scalability and privacy. In this paper, we discuss that scalability and privacy of smart contracts can be enhanced by splitting a contract into two separate contracts: (1) an *off-chain* contract encapsulating functions of the whole contract that involve high-cost computation and/or distinguishable logic that may reveal private information about the participants; (2) an *on-chain* contract encapsulating the remaining low-cost/non-sensitive functions of the whole contract. Replacing the whole smart contract with the pair of on/off-chain contracts enables deploying only the *on-chain* contract onto the blockchain. This also allows saving the computing cost of running the *off-chain* contract by miners in the Ethereum blockchain network while the sensitive information involved in the *off-chain* contract can be hidden from the public.

After generating the pair of on/off-chain contracts from the whole contract, when all the participants are honest, they can reach unanimous agreement on the result of *off-chain* contract, just as if the *off-chain* contract was deployed *on-chain* and executed by miners. However, if there is a dispute of the *off-chain* result, such as a dishonest participant trying to lie about it, it would be necessary to allow honest participants to resolve the dispute *on-chain*. In this paper, we propose an effective mechanism that allows any honest participant to leverage the already deployed *on-chain* contract to create an *on-chain* instance of the *off-chain* contract. The process of creating a new contract from an existing contract can establish a unique connection between the two contracts. Through this connection, only the *off-chain* contract agreed and signed by all the participants can pass the integrity verification and can be created by the *on-chain* contract as a verified instance. Then, only the execution result of the verified instance can enforce the state change of the *on-chain* contract. Thus, honest participants have the ability to redress and penalize any fraudulent or dishonest behavior, which incentivizes all participants to honestly follow the agreed *off-chain* contract. We implement and evaluate the proposed approach in the Ethereum official test network *Kovan* [7] with an example smart contract using *Solidity* [16]. Our implementation demonstrates the effectiveness of the proposed approach of using *on/off-chain* contracts in Ethereum.

IEEE
computer society

II. ON/OFF-CHAIN SMART CONTRACTS

In this section, we first introduce the creation and execution of a general smart contract in Ethereum. We then present the techniques for splitting a given smart contract into a pair of *on/off-chain* contracts and explain how the pair of contracts can be executed in Ethereum.

A. Smart contracts in Ethereum

There are two types of accounts in Ethereum, namely External Owned Accounts (EOAs) controlled by private keys owned by users and Contract Accounts (CAs) assigned to smart contracts. A user of Ethereum should first create a EOA with a pair of keys and then deploy smart contracts from the EOA, resulting in the creation of CAs associated with the smart contracts. A smart contract in Ethereum refers to a piece of program code that usually consists of multiple functions, a few parameters and perhaps some modifiers. After programming a smart contract in a language such as *Solidity* [16], a user can compile the contract to get its *bytecode* and Application Binary Interface (*ABI*) and can send a contract creation transaction to the Ethereum network with *bytecode* and (optional) *ABI*. Upon receiving the transaction, miners will include the *bytecode* into the next block, meaning that a new smart contract has been created, whose CA can be deterministically computed from the address of its creator and a nonce. Each CA can be viewed as a small decentralized server that can act based on the functions in its contract and can store data (e.g., cryptocurrency) allowed by its contract. However, CAs are passive, meaning that execution of any function of deployed smart contracts must be invoked through either transactions sent by EOAs or messages sent from CAs. As a result, the transactions/messages, as well as function inputs inside them, are all recorded by the Ethereum blockchain, which makes the function outputs deterministic because all miners can execute the function with the same inputs and gets the same outputs. It is worth noting that one needs to pay Gas [19] for either deploying a new smart contract or calling a function of existing smart contracts in Ethereum. Gas can be exchanged with Ether, the cryptocurrency used in Ethereum, and Ether can be exchanged with real money.

As discussed above, Ethereum requests each invoked function of deployed *on-chain* contracts to be executed by all the miners for the purpose of getting trustworthy outputs in the decentralized environment. However, this *all-on-chain* execution model of smart contracts may drain the scarce network resources and introduce privacy risks, when the invoked function includes high-cost calculation or sensitive information. Therefore, instead of deploying a whole contract with all function included, we propose to split such problem functions from the whole contract to execute them *off-chain* while still executing the rest of functions *on-chain*. We present the technique of implementing this approach by splitting a given contract into a pair of *on/off-chain* contracts.
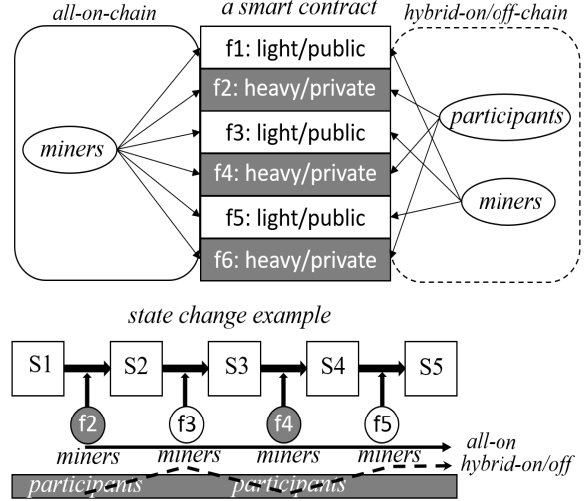

Fig. 1: Comparison of the current *all-on-chain* execution model of smart contracts in Ethereum and the proposed *hybrid-on/off-chain* execution model

B. On/off-chain smart contracts

We broadly classify functions of smart contracts into two categories based on the computing resources spent for executing them and the sensitive information carried by them: (1) *heavy/private* functions involving high-cost calculation or sensitive information and (2) *light/public* functions involving none of these features. For example, Alice and Bob may decide to bet on a private topic between them using cryptocurrency they have, so they draft a simplified betting contract as shown in Algorithm 1, which consists of three functions. They can first make deposits to the contract through *deposit()*, then invoke *reveal()* after a certain temporal threshold to reveal the result and finally reassign the cryptocurrency locked in the contract based on the result by calling *reassign()*. Since both *deposit()* and *reassign()* are simple cryptocurrency transfer functions, they contain neither high-cost calculation nor sensitive information, so they can be allocated to *light/public* functions. In contrast, *reveal()* may contain details of the customized betting rules that are private to the participants and may involve an arbitrary amount of computational cost, hence it should be allocated to a *heavy/private* function. It could be quite customized to allocate functions into the two categories. Here we recommend to allocate all functions of cryptocurrency transfer into *light/public* functions and consider the remaining ones as *heavy/private* functions.

```
contract betting {
    function deposit() payable public participantOnly;
    function reveal() public participantOnly;
    function reassign() public participantOnly;
}
```

Algorithm 1: A simplified betting contract

After all functions of a smart contract has been classified, the *heavy/private* functions and *light/public* functions can be grouped into an *off-chain* contract and an *on-chain* contract,

respectively. Then, only the *on-chain* contract needs to be deployed to be publicly executed by all the miners while the *off-chain* contract can be privately executed by only a small group of interested participants. Ideally, splitting the *off-chain* contract from the whole contract does not affect the state change of the *on-chain* contract. For clarity of illustration, consider an example smart contract in Fig. 1, which consists of three light/public functions (i.e., $c1$, $c3$, $c5$) and three heavy/private functions (i.e., $c2$, $c4$, $c6$). With the *all-on-chain* model, the whole contract consisting of all the six functions is deployed and miners need to execute function $f2$, $f3$, $f4$ and $f5$ to change the state of the deployed contract from $S1$ to $S5$. In contrast, with the *hybrid-on/off-chain* model, only an *on-chain* contract consisting of function $f1$, $f3$ and $f5$ is deployed, so miners only need to execute function $f3$ and $f5$ while $f2$ and $f4$ can be privately executed by interested participants. As long as the participants are honest, they will be able to reach unanimous agreement on the *off-chain* execution results of $f2$ and $f4$ and input these results to turn the contract state from $S1$ to $S2$ and later from $S3$ to $S4$, just as if both $f2$ and $f4$ were executed by miners *on-chain*. However, in case of dishonest participants trying to lie about the *off-chain* execution results, we need an additional mechanism to always allow honest participants to enforce the *true* execution results of the *off-chain* contract. We will introduce this mechanism in the next section.

III. Enforcing off-chain contracts

Our proposed enforcement mechanism is used by honest participants to enforce the results of *off-chain* execution and penalize any dishonest participants.

In the presence of any dishonest participants, an honest participant can simply deploy the *heavy/private* functions back to the blockchain, make them get recomputed by the miners to enforce the state of *on-chain* contract to be changed as expected. However, there are two challenges to make this strategy work as expected. First, the honest participant must prove that the deployed *heavy/private* functions are exactly same as the original ones agreed by all the participants at the beginning. Otherwise, participants can falsify these functions based on their self-interests. Second, after verifying the integrity of the deployed *heavy/private* functions, we need to rebuild a connection between these functions and the already deployed *light/public* functions (i.e., *on-chain* contract). Since the two groups of functions have been separated from a single smart contract, such a connection is necessary to make them re-recognize each other, just as if they were deployed together in a single smart contract at the beginning.

To overcome the aforementioned challenges altogether, in Fig.2, we propose a general mechanism of enforcing *off-chain* contracts in the *hybrid-on/off-chain* execution model of smart contracts, which consists of four stages:

- *Split/generate*: Before any on-chain activity, the *original* contract is first split to two parts that package *light/public* functions and *heavy/private* functions, respectively. Obviously, the two parts themselves usually do not contain any
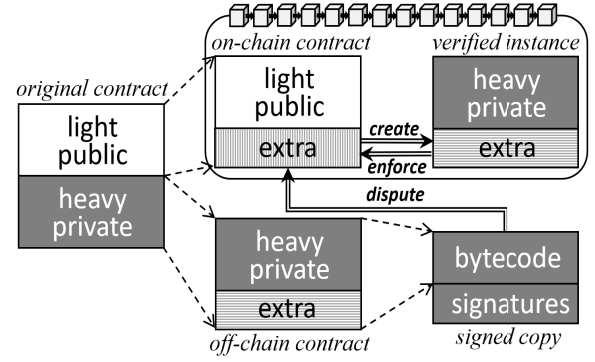

Fig. 2: Two categories of functions in the whole contract are padded with a few extra functions to form the *on-chain* contract and the *off-chain* contract. In the presence of dishonest participants, any honest participant can use the signed copy of the *off-chain* contract to make the *on-chain* contract create a verified instance of the *off-chain* contract and leverage the unique link between the instance and its creator (i.e., *on-chain* contract) to enforce state change of the *on-chain* contract.

function for resolving a dispute, so we need to pad each group of functions with a few extra functions prepared for a dispute. After padding, we have already generated an *on-chain* contract as well as another *off-chain* contract from the whole contract.

- *Deploy/sign*: The *on-chain* contract (i.e., *light/public* functions including the padded extra functions) can then be deployed by any participant to the blockchain while the *off-chain* contract (i.e., *heavy/private* functions plus padded extra functions) needs to be converted into *bytecode* to be signed by all the participants. It is important to note that each participant must obtain a copy of the *off-chain* contract with *signatures* from all the participants before any interaction with the deployed *on-chain* contract can take place. The procedure of generating *signed* copies may easily be implemented through off-chain communication approaches, such as Whisper [18] in Ethereum.

- *Submit/challenge*: When all participants are honest, they can execute computation of the *off-chain* contract by themselves and manually submit the results to the *on-chain* contract to push the state change. This can be implemented by leaving a challenge period after a representative of the participants have submitted the result, during which all other participants can challenge the result with the signed copy of the *off-chain* contract. Obviously, if the representative is honest, there will be no need to challenge the submitted result, so the state of the *on-chain* contract can be successfully changed as expected without revealing any information of the *off-chain* contract and also without requiring miners to perform the computation of *heavy/private* functions.

- *Dispute/resolve*: In the presence of any dishonest participants, such as if the representative violates the agreement, a dispute occurs. To resolve the dispute, during the challenge period, any honest participant can submit the *signed* copy, namely the *bytecode* and *signatures* together, to the *on-chain* contract, where an extra function padded to the *on-*

Betting Rules

Participants Alice and Bob.

Rules:

1. Before time T_0, Alice should deploy *on-chain* contract. Both the participants should keep a signed copy of the *off-chain* contract.
2. Before time T_1, both the participants can make a deposit (1 Ether) to *on-chain* contract through *deposit()* or request a refund through *refundRoundOne()*.
3. Between T_1 and T_2, if either Alice's balance or Bob's balance in *on-chain* contract is not 1 Ether, the participants can request a refund through *refundRoundTwo()*.
4. The time point T_2 is the temporal threshold when the betting result becomes available. After that, between T_2 and T_3, the participants can compute the result *off-chain*. The loser (say Alice) can then call *reassign()* to make *on-chain* contract transfer 2 Ether to winner's account.
5. Finally, if the loser refused to implement step 4, the game will go to *dispute/resolve* stage. After T_3, the winner (say Bob) can call the extra function *deployVerifiedInstance()* with the signed copy at the *on-chain* contract and then call the extra function *returnDisputeResolution()* at the verified instance to enforce the dispute resolution.

TABLE I: Betting rules

chain contract during the *split/generate* stage will be called to verify the *signed* copy through *signatures* and create a verified *on-chain* instance. The deployed verified instance is created through the *bytecode* of the *off-chain* contract, so it is cloned from the *off-chain* contract and consists of all the *heavy/private* functions and padded extra functions in the *off-chain* contract. A participant (may or may not be the one submitting the *signed* copy) can then invoke the *heavy/private* functions within the verified instance to make them be executed by the miners. The execution result will then be the *true* one, which is different from the *false* result submitted by the dishonest participant during the *dispute/resolve* stage. Then, an extra function padded to the *off-chain* contract will be invoked to send the *true* result back to the *on-chain* contract and another extra function padded to the *on-chain* contract will receive the *true* result, enforce the dispute resolution and penalize the dishonest participant. To sum up, in the fourth stage, three extra functions are designed to resolve a dispute:

- *deployVerifiedInstance()*: We design this extra function at the *on-chain* contract to receive the *signed* copy during the challenge period. Once called, this function first verifies all the received *signatures* and then creates an *on-chain* instance for the *bytecode* within a *signed* copy that has passed the verification.
- *returnDisputeResolution()*: We design this extra function at the *off-chain* contract to return the *true* result computed by miners to the *on-chain* contract.
- *enforceDisputeResolution()*: We design this extra function at the *on-chain* contract to enforce the dispute resolution.

With this four-stage mechanism, honest participants always keep the ability to make dishonest participants be verified and monetary penalized, which in turn incentivizes all participants to honestly follow the results of off-chain smart contracts.

Next, we design and implement the proposed approaches using a detailed example.

IV. IMPLEMENTATION

We implement the proposed approaches using an example smart contract for betting between Alice and Bob. The betting rules in the example smart contract are listed in Table I. Note that the presented betting rules do not involve penalty to dishonest participants, however it is straight-forward to revise step 2, 3 and 4 to monetarily penalize any dishonest participant in step 5.

The contracts are programmed with *Solidity* and the *off-chain* signature part is implemented using *JavaScript* with ethereumjs-util [4] package and web3-utils [17] package. All the contracts have been tested over the Ethereum official test network *Kovan* [7].

```solidity
pragma solidity ^0.4.24;

contract onChain {
  // light/public functions
  function deposit() payable public beforeT1
    certifiedparticipantOnly;
  function refundRoundOne() public beforeT1
    certifiedparticipantOnly;
  function refundRoundTwo() public T1toT2
    certifiedparticipantOnly amountNotMet;
  function reassign() public T2toT3
    certifiedparticipantOnly;
  // extra functions
  function deployVerifiedInstance(bytes memory
    _bytecode, uint8 _va, bytes32 _ra, bytes32 _sa,
    uint8 _vb, bytes32 _rb, bytes32 _sb) public
    afterT3 certifiedparticipantOnly amountMet;
  function enforceDisputeResolution(bool _winner)
    external deployedAddrOnly;
}
```

Algorithm 2: *on-chain* contract (interface)

```solidity
pragma solidity ^0.4.24;

// interface
contract onChain {
  function enforceDisputeResolution(bool _winner)
    external;
}

contract offChain {
  modifier certifiedparticipantOnly {...}
  // heavy/private functions
  function reveal() private returns(bool) {...}
  // extra functions
  function returnDisputeResolution(address _addr)
    public certifiedparticipantOnly {
    onChain C_on = onChain(_addr);
    C_on.enforceDisputeResolution(reveal());
  }
}
```

Algorithm 3: *off-chain* contract (simplified)

In rule 1, before T_0, the *on-chain* contract should be deployed and the participants should have obtained *signed* copy of the *off-chain* contract. We show the implementation of the two contracts with *Solidity* in Algorithm 2 and Algorithm 3, respectively. As can be seen, in this example, the *on-chain* contract consists of four light/public functions and two extra functions while the *off-chain* contract consists of one heavy/private function and one extra function. Here we note

that the parameters, constructor and modifiers (e.g., beforeT1, certifiedparticipantOnly) are omitted in Algorithm 2.

```
var web3Utils = require('web3-utils')
var ethUtils = require('ethereumjs-util')
var _bytecode = '0x608060405234801...bab40029'
var h_bytecode = web3Utils.soliditySha3(_code)
var h_bytecode_hex=new Buffer(h_bytecode.slice(2),'hex')

var privkey_a = new Buffer('499...38c', 'hex')
var vrs_a = ethUtils.ecsign(h_bytecode_hex, privkey_a)
var _va = vrs_a.v
var _ra = vrs_a.r.toString('hex')
var _sa = vrs_a.s.toString('hex')

var privkey_b = new Buffer('360...572', 'hex')
var vrs_b = ethUtils.ecsign(h_bytecode_hex, privkey_b)
var _vb = vrs_b.v
var _rb = vrs_b.r.toString('hex')
var _sb = vrs_b.s.toString('hex')
```

Algorithm 4: Signature v-r-s generation

```
contract onChain {
  ...
  address public deployedAddr;

  function deployVerifiedInstance(bytes memory
    _bytecode, uint8 _va, bytes32 _ra, bytes32 _sa,
    uint8 _vb, bytes32 _rb, bytes32 _sb) public
    afterT3 certifiedparticipantOnly amountMet {
    // verify signatures
    bytes32 h_bytecode = keccak256(_bytecode);
    address a = ecrecover(h_bytecode, _va, _ra, _sa);
    address b = ecrecover(h_bytecode, _vb, _rb, _sb);
    require(a == certifiedparticipant[0] && b ==
      certifiedparticipant[1]);
    // create verified instance
    address _addr;
    assembly {
      _addr := create(0, add(_code, 0x20), mload(_code))
    }
    deployedAddr = _addr;
  }
  ...
}
```

Algorithm 5: *deployVerifiedInstance()*

```
contract onChain {
  ...
  function enforceDisputeResolution(bool _winner)
    external deployedAddrOnly {
    accountBalance[participant[0]] = 0;
    accountBalance[participant[1]] = 0;
    if(_winner == true) {
      participant[1].transfer(
      accountBalance[participant[0]] +
      accountBalance[participant[1]]);
    } else {
      participant[0].transfer(
      accountBalance[participant[0]] +
      accountBalance[participant[1]]);
    }
  }
  ...
}
```

Algorithm 6: *enforceDisputeResolution()*

After generating the two contracts, Alice and Bob should transform the *off-chain* contract shown in Algorithm 2 into *bytecode* through tools such as *Remix* or *Truffle* and then use the *JavaScript* program in Algorithm 4 to generate signatures (tuples of *(v,r,s)*) with their account private keys and the *bytecode*. Please note that all the participants should use the

same version of compiler for the purpose of getting same *bytecode*.

In case if Alice or Bob violates the rules before T_2, the game can always be terminated through *refundRoundOne()* or *refundRoundTwo()*. After T_2, suppose Alice loses the game, she should call *reassign()* to admit this failure before T_3 to let the function transfer both winner's deposit and loser's deposit to winner. If *reassign()* is not called before T_3, the winner will find the fact that the loser has violated the rules and a dispute has occurred. Once the dispute happens, after T_3, the winner (say, Bob in this example) should call the *deployVerifiedInstance()* extra function (Algorithm 5) with the *signed* copy. The extra function will first verify the signatures by outputting an address from *bypecode* and *(v,r,s)* and checking whether it is the signer's address. After that, the extra function will create a verified instance from the *bytecode* and also record the address of the created instance in the parameter *deployedAddr*. Thus, by generating the verified instance from the *on-chain* contract, the verified instance can be authorized through the parameter *deployedAddr* because no other contract can have the same address and it is also guaranteed that the verified instance has been agreed by all the participants. Please note that to create a contract from an existing contract with only *bytecode*, we must use the assembly language in *Solidty*.

Finally, the winner (i.e., Bob) can call *returnDisputeResolution()* in the verified instance (see Algorithm 3) with the address of the *on-chain* contract (see Algorithm 2). The input address of the *on-chain* contract will then allow the verified instance to return the dispute resolution to the *on-chain* contract by invoking the extra function *enforceDisputeResolution()* (Algorithm 6), which then enforces both winner's deposit and loser's deposit to be transferred to winner and may also monetarily penalize the dishonest loser if there is a penalty rule. Here, the *deployedAddrOnly* modifier will check *msg.sender* so that only the verified instance with the address same as the one recorded in parameter *deployedAddr* can leverage *enforceDisputeResolution()* to change the state of *on-chain* contract.

Extra function	Gas cost
deployVerifiedInstance()	225082 + *reveal()*
returnDisputeResolution()	37745

TABLE II: Gas cost

We now present the gas cost for dispute resolution. As shown in Table II, in case of a dispute, an honest participant needs to first spend (225082 + cost of *reveal()*) gas to deploy the verified instance through *deployVerifiedInstance()* and then spend 37745 gas to enforce the resolution through *returnDisputeResolution()*. The overall cost is not high when cost of *reveal()* is low. However, if *reveal()* is a heavy function, it should be mandatory for each participant to pay security deposit so that the honest participant paying for dispute resolution can receive compensation from dishonest participants.

11

V. RELATED WORK

Recently, *off-chain* resources of blockchain have been widely studied by researchers to improve the performance of blockchains [2], [5], [8], [9], [10], [13]. Among them, [5] and [10] are most relevant to our work. In [10], smart contracts are implemented using hybrid architectures similar to the *hybrid-on/off-chain* execution model proposed in our work. However, the proposed hybrid architectures in [10] rely on a Trusted Third Party (TTP) as an oracle and thus the architectures are not completely decentralized. In [5], a smart contract system named Arbitrum is developed, where smart contracts are designed to be executed *off-chain*. As the system is specially designed for this purpose, it is hard to generalize the system-level design to existing systems such as Ethereum. In this paper, instead of treating the use of *off-chain* contracts as a system-level design goal, we consider the *hybrid-on/off-chain* computation model as an application-level smart contract design pattern and also as a building block for enhancing blockchain scalability and privacy. Thus, the proposed approach is a plug-and-play solution that is compatible with existing smart contract systems and their time-tested infrastructure and community. In addition, the combination of the proposed approach and other system-level or application-level solutions, such as sharding [1] and zero knowledge proof [6], can further enhance the scalability and privacy of the smart contract systems.

VI. CONCLUSION

In this paper, we propose a general *hybrid-on/off-chain* execution model of smart contracts, which separates the heavy/private functions of a smart contract from the light/public ones to form an *off-chain* contract and enables the *off-chain* contract to be executed only by the interested participants. The proposed approach leads to increased saving of computing resources of the miners and protects sensitive information of a smart contract in the blockchain. To handle disputes caused by any dishonest participants, we propose a mechanism that allows any honest participant to reveal a signed copy of the *off-chain* contract so that a verified instance can be created to make miners enforce the true execution result. Therefore, with the proposed approach, honest participants always have the ability to redress and penalize any fraudulent or dishonest behavior, which incentivizes all participants to honestly follow the agreed *off-chain* contract. Our implementation and evaluation of the proposed approach using an example smart contract demonstrate the effectiveness of the proposed approach of using *on/off-chain* contracts in Ethereum.

REFERENCE

[1] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778*, 2017.

[2] Jacob Eberhardt and Stefan Tai. On or off the blockchain? insights on off-chaining computation and data. In *European Conference on Service-Oriented and Cloud Computing*, pages 3–15. Springer, 2017.

[3] Ethereum market cap. https://coinmarketcap.com/currencies/ethereum/.

[4] ethereumjs-util. https://github.com/ethereumjs/ethereumjs-util.

[5] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium*, pages 1353–1370. USENIX Association, 2018.

[6] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE symposium on security and privacy (SP)*, pages 839–858. IEEE, 2016.

[7] Kovan: Ethereum official testnet. https://kovan.etherscan.io/.

[8] Chao Li and Balaji Palanisamy. Decentralized privacy-preserving transaction scheduling in blockchain-based smart contract platforms. In *High Performance Computing (HiPC), 2018 IEEE 25th International Conference on*. IEEE, 2018.

[9] Chao Li and Balaji Palanisamy. Decentralized release of self-emerging data using smart contracts. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 213–220. IEEE, 2018.

[10] Carlos Molina-Jimenez, Ioannis Sfyrakis, Ellis Solaiman, Irene Ng, Meng Weng Wong, Alexis Chun, and Jon Crowcroft. Implementation of smart contracts using hybrid architectures with on and off–blockchain components. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 83–90. IEEE, 2018.

[11] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[12] Neo. https://neo.org/.

[13] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. *See https://lightning. network/lightning-network-paper. pdf*, 2016.

[14] Smart contracts market research report global forecast to 2023. https://www.marketresearchfuture.com/reports/smart-contracts-market-4588.

[15] State of the dapps. https://www.stateofthedapps.com/.

[16] The solidity contract-oriented programming language. https://github.com/ethereum/solidity.

[17] web3-utils. https://www.npmjs.com/package/web3-utils.

[18] Whisper. https://github.com/ethereum/wiki/wiki/Whisper.

[19] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151, 2014.