Game Theoretic Approach for Secure and Efficient Heavy-Duty Smart Contracts

Pinglan Liu, Wensheng Zhang
Computer Science Department, Iowa State University
{pinglan,wzhang}@iastate.edu

***Abstract*—Smart contracts in the blockchain systems such as Ethereum are usually executed or verified by all nodes, and thus inefficient for heavy-duty computation. This paper addresses the limitation by proposing, implementing, and evaluating a practical and efficient solution based on a game theoretic approach. The solution defines a template of heavy-duty smart contract (HDSC); recruits only a small number of executors to execute heavy-duty tasks; employs a game theoretic scheme to enforce economically-rational executors to individually or collectively perform the execution correctly. Extensive game theoretic analysis has been conducted to show the security and computational efficiency of the solution even in face of collusion among the executors. As a proof of concept, the proposed solution has been implemented and experimented to demonstrate its practicality and compatibility with Ethereum.**

***Index Terms*—Blockchain, Smart Contract, Computation Efficiency, Game theory, Collusion**

I. Introduction

Starting from Bitcoin [1] to Ethereum [2], the functionality of blockchain is completed from pure cryptocurrency to smart contracts that enable general-purpose computation in the verifiable, transparent and non-repudiable manner. However, in most of the currently popular blockchain systems, the smart contracts have to be executed or verified by all nodes, which leads to significant waste of computation and limited scalability, and consequently makes smart contracts more suitable for lightweight but not heavy-duty computation.

Related Works & Limitations: Studies on improving the computational efficiency of blockchain are extensive. The proof-of-stake (PoS) mechanism [3], [4], [5] which selects nodes to be block creators based on random selection, wealth, age, or a combination of these, has been explored as an alternative of the proof-of-work (PoW) which selects block proposers based on highly energy-consuming competitions in solving cryptographic puzzles. A PoS-based system, however, still requires every stakeholder to participate in verifying (which often means repeating) every execution. There are proposals of utilizing the trusted execution environment (TEE) [6], [7], [8] to achieve the community computation instead of having all nodes involved. However, currently prevalent TEE such as Intel SGX is inappropriate for multi-threading and tasks requiring high demand of memory which could make it a bottleneck of enormous computation and validation on blockchain. The sharding-based approach [9], [10], [11] has been proposed to divide a blockchain into multiple subchains for scalability; however, each subchain still needs a large number of members to guarantee security, and hence each smart contract is still repeatedly executed by a large number of nodes. Researchers also proposed the hierarchical approach, for example Thunderella [12], where periodically a leader and a committee are elected to execute and verify transactions (including smart contracts), and all the nodes on the chain are resorted to only when the leader is found not trustable; such approach may not improve the scalability for systems with heavy-duty tasks as there is only a small number of leaders at a time to execute them.

Recently, the on-chain/off-chain hybrid approach [13], [14], [15], [16], [17], [18] has been proposed, which combines off-chain execution of heavy-duty or private tasks with on-chain execution of lightweight and privacy-insensitive tasks. However, the state-of-the-art designs have several limitations, such as difficult-to-satisfy requirements posed to users, no consideration of collusion, and inefficient use of on-chain operations, which may hinder their application in practice. For example, the designs proposed in [14], [15], [16], [17] all assume no collusion among the involved parties, and require the users to find executors off-chain; some designs require the off-chain executors to include at least one being honest [16], or be configured with a TEE [18] or even a TTP (trusted third party) [15]; some designs require the off-chain executors to run secure multiple-party computation protocols [17]. Average blockchain users may find such requirements inconvenient or infeasible. Some of the designs [16] introduce multiple rounds of on-chain challenge-response operations to verify computation results, which could pose high cost to both on-chain nodes and off-chain executors.

Our Contributions: The hybrid approach is promising in improving the efficiency of handling heavy-duty computation in blockchain, but remains impractical or inefficient due to the afore-discussed limitations. This paper aims to address this issue by proposing, implementing, and evaluating a practical and efficient solution based on the approach.

In the proposed solution, a user who has a heavy-duty task simply posts to the blockchain a heavy-duty smart contract (HDSC) that follows our designed template, where the desired number of executors and other optional parameters are specified; then, the user deposits money that should be paid to the honest executors. Once the HDSC gets ready for execution, the executors who are either selected from the blockchain nodes following a default rule or picked by the user, make deposits to be responsible for their executions, execute individually

978-1-7281-4760-4/20/$31.00 ©2020 IEEE

or collectively the computation task embedded in the HDSC, commit the execution results, reveal the results, and finally get rewards (if honestly executing the computation) or lose deposits (otherwise). With a low and adaptable frequency, a trusted mechanism (e.g., the underlying blockchain, available TEE, or other mechanism optionally picked by the user) also executes the HDSC, which serves as a deterrence to executors' misbehavior. A game theoretic scheme is designed to regulate the above operations such that, economically-rational executors have to act honestly or collude only in a *benign* way where computation cost can be saved but the computation result is guaranteed correct.

Compared to the afore-discussed related works, the proposed solution has the following features: (i) *Compatibility* - Users can launch a HDSC in a way almost the same as launching a regular smart contract. They are not required to find executors; if they choose to pick executors themselves, there is no expectation for trusted executors. The proposed solution can run atop a general blockchain system, though Ethereum is assumed throughout the paper. (ii) *Security* - As demonstrated through game theoretic analysis, the proposed solution enforces economically-rational executors to conduct computation honestly and submit correct results, no matter whether they collude or not. (iii) *Efficiency* - The computation is mostly conducted by the selected executors; the more expensive trusted computation mechanism such as the underlying blockchain or available TEE is only used as deterrence with low frequency, which is $\frac{c}{n \cdot d_{executor}}$ where c is the cost for executing the HDSC computation, n is the number of executors and $d_{executor}$ is the deposit made by each executor; hence, the more executors are selected, the low frequency for using the trusted computation mechanism. Furthermore, the proposed solution encourages *benign collusion* where executors collude by having only one of them to honestly conduct the computation, which can further reduce the computation cost. (iv) *Flexibility* - The proposed solution tolerates dynamics in the blockchain, such as churning and failure of executors. Also, users are given the freedom to determine the operational parameters for each HDSC.

Organization: In the following, Section II presents the preliminaries. Section III provides an overview of our proposed solution. Section IV presents and analyzes our proposed game theoretic schemes. Section V reports an implementation of the proposed schema and discusses the evaluation results. Finally, Section VI concludes the paper.

II. Preliminaries

System Model: We consider a blockchain system that supports verifiable, transparent and non-repudiable general-purpose computation submitted as smart contracts. We classify the smart contracts into *heavy-duty smart contracts (HDSCs)* and regular smart contracts. A HDSC demands a heavy computation, and thus is ideal to be executed by only a small subset (instead of all) of the nodes in the system. Whether a smart contract is heavy-duty or regular can be determined by a certain policy. For example, the gas upper bound of a block in Ethereum can be set to x; hence, a smart contract that may consume more than $\frac{x}{y}$ gas can be classified as heavy-duty if a block is expected to store more than y transactions. We also assume the computation in each HDSC is *deterministic*; hence, the result of the computation should be the same if it is honestly executed by different parties.

Security Assumptions: We assume the blockchain system itself is *secure*, and has the following specific features: (i) The nodes who add blocks to the blockchain are selected randomly. Particulary, in a proof-of-work based system, the probability for a node to be selected is proportional to the percentage of computation power that it owns; in a proof-of-stake based system, the probability is proportional to the percentage of stake that it holds. (ii) There exists a bound α such that, a block that has been confirmed by α blocks linearly following it is *stable* (or *irreversible* with an overwhelming probability).

We assume the existence of a trusted computing mechanism. As a baseline, posting a computation task as a regular smart contract and calling on all nodes to execute or verify the result can serve the purpose. For efficiency, it is more desirable that the trusted computing mechanism only involves a subset of the nodes (e.g., through the sharding-based or hierarchical approach) or some special nodes (e.g., nodes equipped with trusted computation hardware).

We assume each node in the system is financially rational and aims to maximize its own payoff.

Also, we assume a misbehaving node that is selected to execute a HDSC may collude with others, and may attempt to: (i) free-ride via sharing other's execution result without really conducting the required execution; (ii) make up a result without real execution; (iii) accuse other executors who are innocent to obtain a reward that it does not deserve.

Design Goals: We aim to design a solution that meets the following goals: (i) *Security* - Each HDSC must be correctly executed. Particularly, if an incorrect result is submitted by an executor, it can be detected and rectified. (ii) *Efficiency* - The execution of each HDSC should involve as few nodes as possible and incur as low cost as possible. Although low delay is desirable, we assume the computation in each HDSC is delay insensitive. (iii) *Flexibility* - The solution should be adaptive to system dynamics, including failures and churning of nodes. (iv) *Compatibility* - The solution can run atop a general blockchain systems, e.g., Ethereum.

III. Solution Overview

In our proposed solution, a heavy-duty smart contract (HDSC) is processed according to the following framework.

Posting a New Smart Contract: Like a regular smart contract, a HDSC should be posted to the blockchain before execution. The HDSC owner broadcasts a transaction containing the HDSC, and waits for it to be posted and stabilized. Once it is posted stably, the owner makes a deposit to the account of the HDSC, used later to pay the executors. To facilitate processing, a HDSC contains the following:

- A flag indicating this smart contract as a HDSC.

- System parameters of this HDSC, including: n, the number of desired executors; the maximum time period for executing and committing the execution results; the maximum time period for revealing the execution results after the commitments have been posted; w, the wage paid to each executor who provides the correct result; $d_{executor}$, the deposit that should be made by each executor; optionally a list of executors picked by the HDSC owner.
- Code for the computation task that should be executed.
- Current state of the HDSC, which is one of the following: *initial*, when the HDSC is posted but its owner has not made deposit; *unready*, when the owner has made the deposit, but the HDSC is waiting for inputs needed for execution; *ready*, when the HDSC has collected needed inputs; *result-committed*, when the desired executors have posted their commitments for results or the maximum time period for execution and commitment has expired; *result-revealed*, when the executors who had committed have revealed their results, or the maximum time period for revelation has expired; *complete*, when the HDSC has been completely handled.
- Functions called to process the HDSC: *owner-deposit*, called by the owner to make deposit; *provide-input*, called by a user to provide an input; *commit-result*, called by an executor to commit its execution result and meanwhile make a deposit to be responsible for the execution; *reveal-result*, called by an executor to reveal its execution result; *report-collusion*, called by a user to report a collusion and meanwhile make a deposit to be responsible for the reporting; *resolve-dispute*, called to launch a trusted execution if there is a collusion report or a discrepancy in the revealed results; *distribute-fund*, called after the results of the executors' computation and/or the launched trusted executions have completed, to distribute the deposits to involved parties (owner, executors and collusion reporters). Note that, other than the *provide-input* function which is computation-specific, the rest of the above functions are generic and independent of the computation embedded in the HDSC.

Providing Inputs: For a HDSC that requires one or more inputs, users should call the *provide-input* function to provide inputs. Once a required number of inputs have been provided, or a certain pre-specified time period for providing inputs has expired, the HDSC transits to the *ready* state.

Executing a Ready HDSC and Committing Results: For a HDSC that transits to the *ready* state at block b_0, its executors should start executing the embedded computation code; by default its executors are the miners of blocks $b_0+1, \cdots, b_0+n$. After an executor has executed the code, it does not immediately disclose the result, to prevent other executors from copying the result. Instead, the executor calls the *commit-result* function to post a cryptographic commitment, e.g., the well-known Pedersen's commitment [19], of the result. With this function, the executor also makes a deposit to be responsible for the result.

Revealing Results: After all executors have posted commitments or the specified time period for computation and commitment has expired, each of the executors who have committed should call the *reveal-result* function to open its commitment and thus reveal the computation result. Note that the number of desired executors and the actual executors who eventually reveal their results can be different.

Reporting Collusion: Executors may collude to save computation cost. In the proposed system, an executor or other users are encouraged to report collusion. During the time periods for result commitment and revelation, a collusion reporter can call the *report-collusion* function to report collusion and meanwhile make a deposit to be responsible for the report.

After-revelation Processing: After all committed results have been revealed or the period for result revelation has expired, the results are compared. If all of the results are the same and no collusion report is received, the executors are all marked as *honest*. Otherwise, the *resolve-dispute* function is called to launch a trusted computation; based on the result of the trusted computation, each executor can be marked as either *honest* or *dishonest*. Finally, the *distribute-fund* function is called to distribute the deposits to the owner and the honest executors.

IV. Game Theoretic Schemes And Analysis

To regulate the operations in the afore-described framework, we propose game theoretic schemes called *outsourcing contracts*. The schemes specify the interactions between the HDSC owner and the HDSC executors, reward the honest executors, punish the misbehaving executors, and encourage executors to report collusion. In this section, we first present two outsourcing contracts in Section IV-A. The first contract only reactively triggers the trusted computation mechanism when discrepancy exists in executors' results or a collusion is reported; the second contract further proactively triggers the trusted computation mechanism with a certain probability $p_{fallback}$, to deter the executors' misbehavior. To analyze the security and efficiency of these two contracts, we conduct the following game theoretic analysis:

- In Section IV-B, we classify collusion into two categories - benign collusion and malignant collusion, and model the executors' collusion into two generic contracts.
- In Section IV-C, we analyze the games induced from the First Outsourcing contract and the collusion contracts, and show the existence of positive probability for malignant collusion to succeed; hence, the First Outsourcing contract is defective in security.
- In Section IV-D, we analyze the games induced from the Second Outsourcing contract and the collusion contracts, and show that, economically-rational executors will not conduct a malignant collusion, as long as $p_{fallback} \geq \frac{c}{n \cdot d_{executor}}$ in the Second Outsourcing contract, where c is the cost for computation, n is the number of executors and $d_{executor}$ is each executor's deposit.

The analysis demonstrates the necessity of proactively launching the trusted computation mechanism as deterrence to ex-

ecutors' misbehavior. It also reveals the relation between n, $d_{executor}$ and $p_{fallback}$; that is, the frequency for using the deterrence decreases as the number of executors or the amount of deposit an executor is required to make increases. So the frequency can be low, and thus the proposed Second Outsourcing contract can be both secure and efficient. Due to space limit, the proofs are skipped here, but can be found at [20].

A. Outsourcing Contracts

We propose two *Outsourcing contracts* in the following.

1) First Outsourcing Contract: The contract contains the following phases:

Phase I - Initial Depositing. The HDSC owner makes a deposit of $d_{owner} = n \cdot w$. Here, w is the wage paid to each honest executor and should be greater than c to encourage executors to participate. Each executor also makes a deposit of $d_{executor}$, and the deposit should be greater than the cost for a trusted computation mechanism to execute the HDSC, which is denoted as c_{full}.

Phase II - Submission of Execution Results. This phase starts after all executors have made deposits or a specified time duration for depositing has passed; it contains the following steps. *Stage (i) - Commitment of Results*. In this stage, every executor submits a cryptographic commitment of its execution result. *Stage (ii) - Revelation of Results*. After Stage (i) complete or a specified time duration for commitment has expired, every executor opens its commitment posted during Stage (i) to reveal its execution result.

Phase III - Anonymous Reporting of Collusion. This phase starts after all the executors who have committed have revealed their results or a specified time duration for result revelation has expired. An executor may choose to report a collusion anonymously by making a deposit $d_{report} = c_{full}$. The identity of this traitor of collusion is implicitly represented by the ID of the account that launches the reporting. Note that, the ID should be different from and not linkable to the ID of the executor in the Outsourcing contract, so that a traitor is not linkable to any executor.

Phase IV - Dispute Resolution. This phase starts after the time period specified for Phase III expires. If the results revealed by all executors are not the same, a trusted execution is launched to obtain the correct execution result, and the cost is paid from the deposits made by the executors. Otherwise (i.e., all the revealed results are the same), a trusted execution is still launched if there is one or more anonymous reporting of collusion; in this case, the cost for trusted execution is paid from the deposits made by the reporters.

Phase V - Distribution of Deposits. There are two cases: *Case (i) - Executors report different results.* Each collusion reporter takes back its deposit. The executors who are found submitting the correct result are marked as honest, while the others are marked as dishonest. If all executors are dishonest, the remaining deposits are taken by the HDSC owner; otherwise, the deposits are evenly taken by the honest executors. *Case (ii) - Executors report the same result.* If the trusted execution has not been launched or the executors' result is found correct by the trusted execution, the remaining deposits are evenly taken by all the executors; otherwise, the remaining deposits are evenly taken by the collusion reporters.

2) Second Outsourcing Contract: As to be demonstrated later, the First Outsourcing contract cannot prevent the executors from submitting an incorrect execution result. Hence, we further propose *the Second Outsourcing contract*. This contract is the same as the First Outsourcing contract, except for the following added to Phase III: With probability $p_{fallback}$, which is a system parameter, the HDSC owner pretends as an anonymous reporting executor by also making a deposit $d_{report} = c_{full}$. *As to be elaborated later, the probability $p_{fallback}$ is a function of n. Since the expected executors may not all be able to completely participate in the execution, the n used in determining $p_{fallback}$ refer to the number of actual executors who completely participate in the computation. This way, churning and failure of executors can be tolerated.*

B. Contracts among Executors

Economically-selfish executors may collude to save their computation cost. Based on the consequence of collusion, we classify a collusion as either *benign* or *malignant*: If a collusion results in a *correct* result submitted by all the executors participating the collusion, we call it a *benign collusion*. If a collusion results in an *incorrect* result submitted by all the executors participating the collusion, we call it a *malignant collusion*. A benign collusion may save the executors' cost but does not lead to incorrect execution result. Hence, we only target at preventing the malignant collusion.

Based on the way that executors interact with each other in a collusion, we further describe their behaviors and identify two collusion types: *Type-I collusion* - one or more executors provides a result, which is promised as correct, to other executors; in return, the latter offer to the former a reward, which is smaller than the cost they would pay if directly do the computation. *Type-II collusion* - one or more executors proposes a result, which may or may not be correct, and ask other executors to submit the same result; to ensure the latter collude, the former promise to reward a *bribe* to the latter if they do, and meanwhile the latter are required to make deposit and will lose the deposits if they do not.

In the following, we describe the generic contracts for these two types of collusion.

1) Type-I Collusion Contract: We call an executor as leader (denoted as LDR) if it promises to execute the HDSC and shares the result to the others. There could be multiple leaders; for simplicity, we assume there is only one single leader. We call the other executors participating in the collusion as followers (denoted as FLR each). The collusion contact consists of the following three phases, where Phases I and II of this contract should happen after Phase I and before Phase II of the Outsourcing contracts, while Phase III of this contract should happen after Phase V of the Outsourcing contracts.

Phase I - Initial Depositing. The LDR makes a deposit of d_{LDR}. Each FLR also makes a deposit of d_{FLR}, where

$d_{FLR} \leq c$; i.e., the deposit, which will be rewarded to the leader if the provided result is correct, should be no greater than the cost for honestly executing the HDSC by itself.

Phase II - Preparing the Execution Result. This phase includes the following three steps. *Step (i) - LDR Preparing the Result*. The LDR honestly executes the HDSC to get the correct result r with a probability of p_{LDR}; or it makes up an arbitrary result r' otherwise. *Step (ii) - LDR Sharing the Result to Followers*. The LDR shares its prepared result, which is either r or r', to the followers. *Step (iii) - Followers preparing their Results*. For each FLR, it honestly executes the HDSC to get the correct result with a probability of p_{FLR}; or it directly uses the result shared by the LDR.

Phase III - Distribution of Deposits. If the result shared by the LDR is accepted as correct in the Outsourcing contract, the LDR takes all the deposits; otherwise, the deposits are evenly distributed to the followers.

In this contract, the leader is supposed to provide the correct result. If the leader does, all the followers should collectively reward the leader by offering their deposits to the leader. This way, only one computation is conducted by the leader and all the executor share the cost of the computation, instead of every executor repeating the same computation for multiple times; hence, their overall computation cost is reduced.

However, if the leader provides an incorrect result, the followers who use the result can be harmed. That is, if the incorrect result is detected in the Outsourcing contract, every follower that uses the result will lose deposit $d_{executor}$. To protect the followers from such loss, it is economically-rational to set the leader's deposit as follows:

$$d_{LDR} = (n-1) \cdot d_{executor}, \tag{1}$$

This way, if the leader provides an incorrect result and this misconduct is detected, the leader should pay the loss of every follower in this collusion contract. We call a Type-I Collusion contract a *Fairly-responsible Type-I Collusion contract* if d_{LDR} is set as in Equation (1).

Also, a Type-I Collusion can be benign, if the leader provides the correct result, or malignant, if the leader provides an incorrect result and all the followers submit the same result.

2) Type-II Collusion Contract: Similar to Type-I collusion, we call the executor who proposes an execution result as leader (denoted as LDR), call the others in the collusion as followers (denoted as FLR each), and assume there is only one leader for simplicity. The leader makes up an execution result r', and promises a reward to the followers as long as they all use this result to submit and the result is accepted as correct in the Outsourcing contract. The contract consists of three phases, where also similar to the Type-I Collusion Contract, Phases I and II of this contract should happen after Phase I and before Phase II of the Outsourcing contract, while Phase III of this contract should happen after Phase V of the Outsourcing contract.

Phase I - Initial Depositing. The LDR makes a deposit of $d_{LDR} + (n-1) \cdot b$, and each FLR makes a deposit of d_{FLR}. Here, b represents the amount of a bribe that LDR promises to reward each FLR who follows the collusion.

Phase II - Preparing the Execution Result. This phase includes the following steps. *Step (i) - LDR Sharing the Result to Followers*. The LDR makes up a execution result r' and shares it to the followers. *Step (ii) - Followers preparing their Results*. For each follower FLR, it honestly executes the HDSC to get and then report the correct result with a probability of p_{FLR}; or it directly uses the result shared by the LDR otherwise.

Phase III - Distribution of Deposits. There are the following different cases: *Case (i) - At least one executor is found betraying the collusion by reporting a result different from r'*. There are different ways to distribute the deposits, but all share the following: each executor betraying the collusion loses its deposit in the collusion contract; each executor who does not betray can take back its deposit, and further obtain the bribe if it is a follower. *Case (ii) - All executors report r', and the result is accepted as correct.* The LDR takes d_{LDR}, and every FLR takes $d_{FLR} + b$. *Case (iii) - All executors report r', but the collusion is detected (because there is reporting of collusion).* No one can be identified as traitor of the collusion due to the anonymity in reporting. Hence, there are two approaches to distributing the deposits. In the first approach, the deposits are distributed as in the previous case; in the second approach, none of the executors is funded from the deposits. It is straightforward to see that, when the first approach is taken, it must not be a Nash Equilibrium when no executor reporting the collusion because any executor who betrays can obtain higher utility. Hence, here we only consider that the second approach is adopted.

As we can see, a Type-II collusion is malignant if it succeeds, because it will lead to an incorrect execution result being accepted.

C. Analysis I: First Outsourcing Contract v.s. Collusion

In this subsection, we analyze the games induced by the First Outsourcing contract and the two types of collusion, respectively. The purpose is to show that, with the First Outsourcing contract, the game can terminate at malignant collusion.

1) First Outsourcing Contract v.s. Type-I Collusion: Figure 1 illustrates Game 1, which is induced by the First Outsourcing contract and the Type-I Collusion contract. Note that, we do not show the branches with collusion reporting because, in the Type-I collusion the strategy of reporting is dominated by the strategy of checking and submitting the correct result.

In Game 1, there is no pure dominating strategy for the LDR or each FLR. When the LDR submits r, no matter what actions the followers take, it always get the rewards of $w-c+(n-1)d_{FLR}$. When the LDR submits r', it can get either $-d_{LDR}-d_{executor}$ or $w+(n-1)d_{FLR}$ depending whether there is follower honestly computes and submits r. Since $w+(n-1)d_{FLR} > w-c+(n-1)d_{FLR} > -d_{LDR}-d_{executor}$, the LDR would play mixed strategy instead of pure strategy

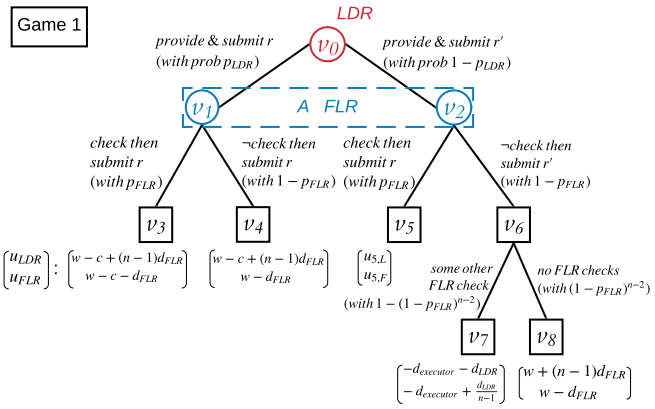


Figure 1: The game induced by the First Outsourcing contract and the Type-I Collusion contract. Here, u_{LDR} and u_{FLR} denote the utility functions of LDR and FLR, respectively; n_0 is the number of executors who compute the result and submit r; n_1 is the number of executors who submit the result r' received from LDR; also, $u_{5,L} = -d_{executor} - d_{LDR}$ and $u_{5,F} = w - c + \frac{d_{LDR}}{n-1} + \frac{n_1 d_{executor} - c_{full}}{n_0}$.

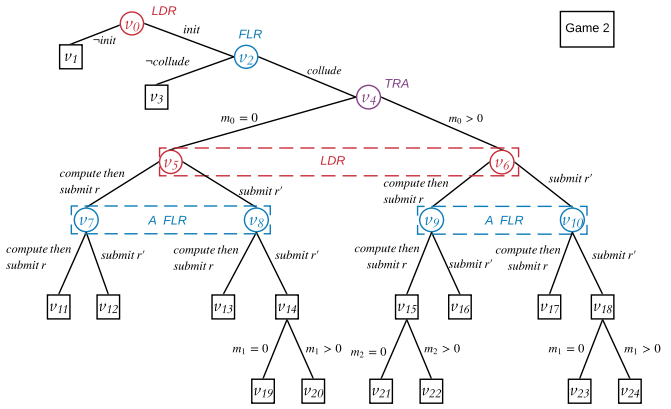


Figure 2: The game induced by the First Outsourcing Contract and the Type-II Collusion.

to maximize its profit. Similarly, for each FLR, a mixed strategy makes more profits than a pure strategy. The following Lemma 1 gives the Nash Equilibrium of Game 1, and the positive probability that the game terminates at a malignant collusion.

Lemma 1: A mixed strategy $\alpha = (\alpha_{LDR}, \alpha_{FLR}) = ((p_{LDR}r, (1-p_{LDR})r'), (p_{FLR}r, (1-p_{FLR})r'))$ is a Nash Equilibrium of Game 1 in Figure 1 where

$$(1-p_{FLR})^{n-1} = 1 - \frac{c}{w + d_{executor} + d_{LDR} + (n-1)d_{FLR}},$$

$$p_{LDR} = 1 - \frac{c}{w + \frac{n d_{executor} - c_{full}}{n_1} - (1-p_{FLR})^{n-2}(w + d_{executor} - d_{FLR} - \frac{d_{LDR}}{n-1})}.$$

Also, both p_{FLR} and p_{LDR} are in $(0,1)$; hence, there is a positive probability $(1-p_{LDR}) \cdot (1-p_{FLR})$ that Game 1 terminates at a malignant collusion.

2) First Outsourcing Contract v.s. Type-II Collusion: Figure 2 illustrates Game 2, which is induced by the First Outsourcing contract and the Type-II Collusion contract, and Table I shows the strategies and corresponding utility functions for the LDR and each FLRs in Game 2.

The LDR can either initiate a collusion or not. If a collusion is initialized, each FLR can either collude or not. Once a collusion is set up, the LDR and each FLR can report the collusion anonymously or not. The LDR can submit either a correct computation result r or a coordinated fake result r'. Also each FLR can either compute honestly and submit r or submit r' provided by the LDR.

Table I shows the normal form representation of the game. The LDR has five strategies as $\neg init$, $init \cdot r \cdot \neg report$, $init \cdot r' \cdot \neg report$, $init \cdot r \cdot report$ and $init \cdot r' \cdot report$. Each FLR has five strategies as $\neg collude$, $collude \cdot r \cdot \neg report$, $collude \cdot r' \cdot \neg report$, $collude \cdot r \cdot report$ and $collude \cdot r' \cdot report$. Each collum corresponds to one strategy of the LDR, and each row corresponds to one strategy of each FLR.

Table I also presents the utilities of the LDR and each FLR under each strategy profile in the corresponding cell, where the first row in the cell is the utility for the LDR and the second row is for the FLR. The following notations are introduced to simplify the presentation:

- $z = w - c$, $d_e = d_{executor}$;
- $d_l = d_{LDR} + (n-1)b$, $d_f = d_{FLR}$;
- $b_1 = \frac{m_2 d_{executor} - c_{full}}{m_1}$; $b_2 = \frac{m_2 d_{executor} - c_{full}}{m_0}$;
- $b_3 = \frac{c_{full}}{m_0}$; $b_4 = -(m_2 - 1)b + \frac{m_1 d_f}{m_2}$;
- $b_5 = \frac{d_l + (m_1 - 1)d_f}{m_2}$; $b_6 = b + \frac{m_1 d_f}{m_2}$;
- m_0 is the number of executors reporting collusion;
- m_1 is the number of executors submitting r;
- m_2 is the number of executors submitting r'.

Note that, the value of m_1 and m_2 may affect the payoffs under some strategies; in those cases, the cell is divided further divided into multiple columns. The Nash Equilibrium with the maximum payoffs is in bold.

Based on Table I, we have the following Lemma 2. Intuitively, the Lemma states that, under certain configurations of the collusion contract that can be controlled by the executors, Game 2 can terminate at a equilibrium where the executors succeed in a malignant collusion.

Lemma 2: When $d_{LDR} \geq (n-1)d_{executor}$, $d_{FLR} \geq (n-1)d_{executor}$ and $(n-1)b < c$, strategy profile $\alpha = (\alpha_{LDR}, \alpha_{FLR}) = (init \cdot r' \cdot \neg report, collude \cdot r' \cdot \neg report)$ is a Nash Equilibrium of Game 2. With this strategy profile, all the executors choose to collude in reporting a faked result r' and not to report it; together with the First Outsourcing constract, this results in a malignant collusion.

3) Summary: Based on the above analysis of games induced from the First Outsourcing Contract, particularly the Lemmas 1 and 2, we have the following Theorem 1.

Theorem 1: If a HDSC owner interacts with executors based on the First Outsourcing contract, which only launches trusted computation reactively when executors report different results or report collusion, there is positive probability that the executors succeed in a malignant collusion.

Table I: First Outsourcing Contract v.s. Type-II Collusion: Strategies and Utilities

$FLR \setminus LDR$	$\neg init$	$init \cdot r \cdot \neg report$	$init \cdot r' \cdot \neg report$	$init \cdot r \cdot report$	$init \cdot r' \cdot report$
$\neg collude$	z z	z z	z z	z z	z z
$collude \cdot r \cdot \neg report$	z z	$m_2 = 0$: $z - d_l$ $z - d_f$ $m_2 > 0$: $z - d_l + b_1$ $z - d_f + b_1$	$-d_e + b_4$ $z - d_f + b_1$	$m_2 = 0$: $z - d_l - b_3$ $z - d_f$ $m_2 > 0$: $z - d_l + b_1$ $z - d_f + b_1$	$-d_e + b_4$ $z - d_f + b_1$
$collude \cdot r' \cdot \neg report$	z z	$z - d_l + b_1$ $-d_e + b_5$	$m_1 = 0$: $\mathbf{w} - (\mathbf{n-1})\mathbf{b}$ $\mathbf{w} + \mathbf{b}$ $m_1 > 0$: $-d_e + b_4$ $-d_e + b_6$	$z - d_l + b_1$ $-d_e + b_5$	$m_1 = 0$: $-d_l - d_e + b_2$ $-d_f - d_e$ $m_1 > 0$: $-d_e + b_4$ $-d_e + b_6$
$collude \cdot r \cdot report$	z z	$m_2 = 0$: $z - d_l$ $z - d_f - b_3$ $m_2 > 0$: $z - d_l + b_1$ $z - d_f + b_1$	$-d_e + b_4$ $z - d_f + b_1$	$m_2 = 0$: $z - d_l - b_3$ $z - d_f - b_3$ $m_2 > 0$: $z - d_l + b_1$ $z - d_f + b_1$	$-d_e + b_4$ $z - d_f + b_1$
$collude \cdot r' \cdot report$	z z	$z - d_l + b_1$ $-d_e + b_5$	$m_1 = 0$: $-d_l - d_e$ $-d_f - d_e + b_2$ $m_1 > 0$: $-d_e + b_4$ $-d_e + b_6$	$z - d_l + b_1$ $-d_e + b_5$	$m_1 = 0$: $-d_l - d_e + b_2$ $-d_f - d_e + b_2$ $m_1 > 0$: $-d_e + b_4$ $-d_e + b_6$

D. Analysis II: Second Outsourcing Contract v.s. Collusion

In this subsection, we first analyze the games induced by the Second Outsourcing contract versus the type-I and type-II Collusion respectively. Then, we summarize the analysis and present our conclusion that, based on the Second Outsourcing contract which launches a trusted computation proactively with a proper probability, the executors will not choose to conduct a malignant collusion, as the collusion will not maximize their utilities.

1) Second Outsourcing Contract v.s. Type-I Collusion: Figure 3 illustrates Game 3 induced by the Second Outsourcing contract and Type-I Collusion. Note that, we skip the branches with collusion reporting as the strategy of collusion reporting is dominated by the strategy of checking and submitting the correct result.

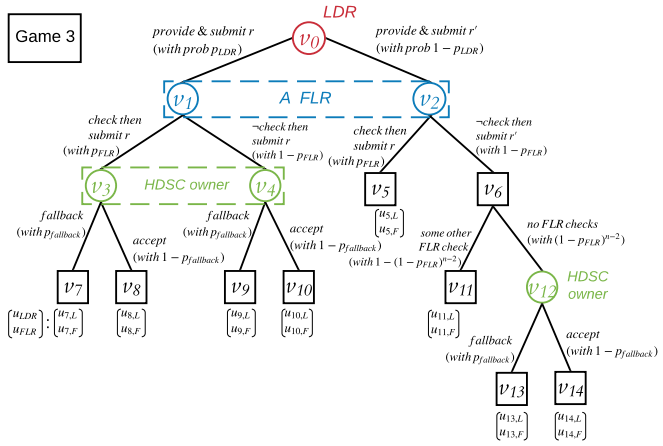


Figure 3: The game induced by the Second Outsourcing Contract and the Type-I Collusion. The utilities for the LDR and each FLR under each strategy profile (i.e., in each branch) is listed as follows: $u_{5,L} = -d_{executor} - d_{LDR}$, $u_{5,F} = w - c + \frac{d_{LDR}}{n-1} + \frac{n_1 d_{executor} - c_{full}}{n_0}$, $u_{7,L} = w - c + (n-1)d_{FLR}$, $u_{7,F} = w - c - d_{FLR}$, $u_{8,L} = u_{7,L}$, $u_{8,F} = u_{7,F}$, $u_{9,L} = u_{7,L}$, $u_{9,F} = w - d_{FLR}$, $u_{10,L} = u_{7,L}$, $u_{10,F} = u_{9,F}$, $u_{11,L} = u_{5,L}$, $u_{11,F} = -d_{executor} + \frac{d_{LDR}}{n-1}$, $u_{13,L} = u_{5,L}$, $u_{13,F} = u_{11,F}$, $u_{14,L} = w + (n-1)d_{FLR}$, and $u_{14,F} = w - d_{FLR}$.

Analyzing the game, we have the following Lemma 3. Intuitively, the lemma shows that, when the HDSC owner has a proper probability to launch a trusted computation proactively, Game 3 will not terminate at a strategy profile in which all the executors collude to submit an incorrect result and not to report the collusion. Hence, the game can only terminate at a state where there is no collusion, or the executors report different results, or some executors report the collusion; none of the cases will make an incorrect result to be accepted.

Lemma 3: As long as $p_{fallback} \geq \frac{c}{d_{executor}}$, Game 3 will not terminate at state v_{13} or v_{14}. That is, rational executors will not collaborate to conduct a malignant collusion.

Though the above Lemma 3 shows that malignant Type-I collusion can be prevented, the frequency for proactive launching of trusted computing needs to be $\frac{c}{d_{executor}}$. However, the following Corollary shows that, a *fairly-responsible Type-I collusion* can be prevented from being malignant with a lower and more scalable frequency for proactive launching of trusted computing. Note that, a fairly-responsible Type-I collusion is more restrictive than a general Type-I collusion by requiring $d_{LDR} \geq (n-1)d_{executor}$; but as explained in Section IV-B1, this additional requirement is economically-rational in practice to protect the followers in the collusion contract against a misbehaving leader who does not keep the promise of providing a correct result.

Corollary 2: For a fairly-responsible Type-I collusion (i.e., $d_{LDR} \geq (n-1)d_{executor}$), as long as $p_{fallback} \geq \frac{c}{nd_{executor}}$, Game 3 will not terminate at state v_{13} or v_{14}. That is, rational executors will not collaborate to conduct a malignant collusion.

2) Second Outsourcing Contract v.s. Type-II Collusion: Figure 4 illustrates Game 4 that is induced by the Second Outsourcing contract and the Type-II Collusion.

Analyzing Game 4, we have Lemma 4 as follows.

Lemma 4: As long as $p_{fallback} \geq \frac{c}{(n-1)\cdot d_{executor}}$, the game shown by Figure 4 will not terminate at state v_{15} or v_{16}.

Note that, in each of the terminal states other than v_{16}, either the type-II collusion is not launched or it not successful (either because the parties submit different results or some parties report the collusion). Hence, according to Lemma 4, as long as $p_{fallback} \geq \frac{c}{(n-1)\cdot d_{executor}}$, a type-II collusion cannot succeed.

3) A Special Case: When all selected executors share the same interest (e.g., they belong to the same miner or the same mining pool), with the Second Outsourcing Contract, we have the following Lemma 5.

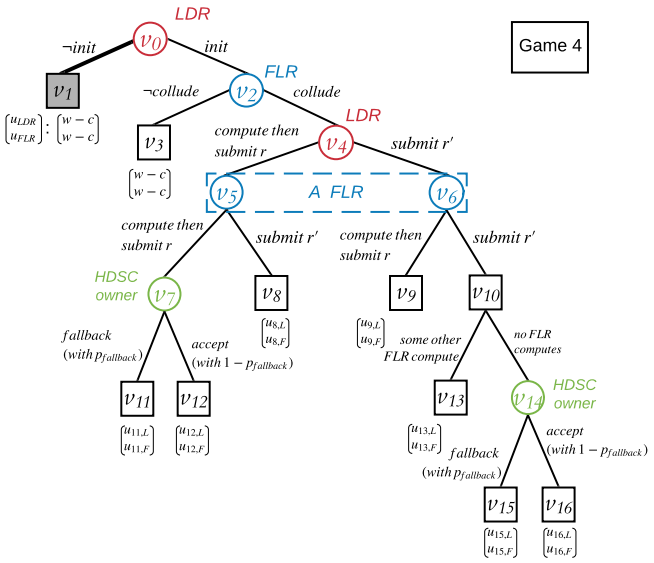


Figure 4: The game induced by the Second Outsourcing contract and the Type-II Collusion. Bold edges indicate the actions that the executors will play in the unique sequential equilibrium. The reachable terminal node of the game is in grey. The utility of LDR and FLR in each branch is listed as follows. $u_{8,L} = z - d_l + b_1$, $u_{8,F} = -d_e + b_5$, $u_{9,L} = -d_e + b_4$, $u_{9,F} = z - d_f + b_1$, $u_{11,L} = u_{8,L}$, $u_{11,F} = u_{9,F}$, $u_{12,L} = u_{8,L}$, $u_{12,F} = u_{9,F}$, $u_{13,L} = -d_e + b_4$, $u_{13,F} = -d_e + b_6$, $u_{15,L} = -d_e - d_l$, $u_{15,F} = -d_e - d_f$, $u_{16,L} = w - (n-1)b$ and $u_{16,F} = w + b$.

Lemma 5: As long as $p_{fallback} > \frac{c}{nd_{executor}}$, the best strategy for all executors with the same interest is to have one executor honestly compute result and all executors submitted the computed result and receive a payoff $n \cdot w - c$ as a group.

4) Summary: Based on the above analysis of games induced from the Second Outsourcing contract and a fairly-responsible Type-I collusion or a Type-II collusion, particularly the Corollary 2, Lemma 4 and Lemma 5, we have the following Theorem 3.

Theorem 3: If a HDSC owner interacts with executors based on the Second Outsourcing contract and the executors' potential collusion is either a fairly-responsible Type-I collusion or a Type-II collusion, economically-rational executors will not conduct a malignant collusion as long as $p_{fallback} \geq \frac{c}{(n-1)d_{executor}}$ in the Second Outsourcing contract.

V. Implementation and Evaluations

As a proof of concept, we implement the proposed system atop Ethereum. Specifically, we implement in Solidity a HDSC template as presented in Section III and the Second Outsourcing contract proposed in Section IV; we develop in Javascript a program which can be called by the Ethereum official Go implementation Geth to handle HDSCs in the way as we propose.

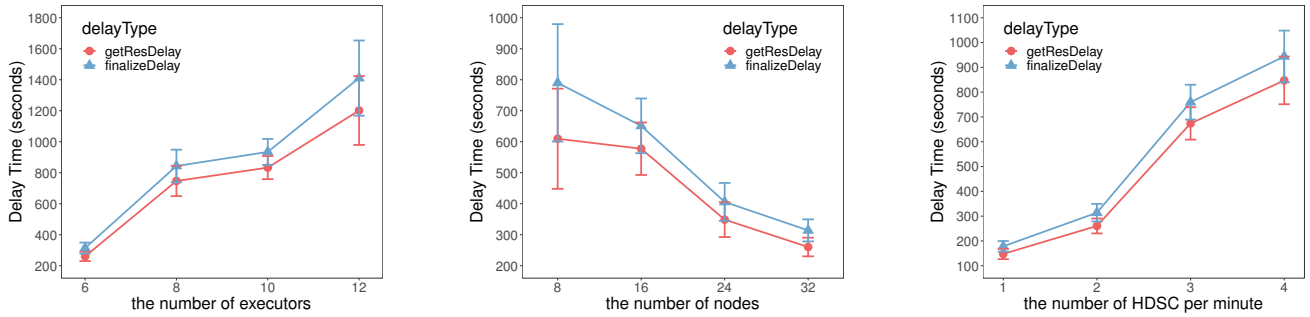
To test and evaluate the system, we rent AWS EC2 instances, each running Ubuntu Server 18.04 LTS with 8GB RAM, to set up a small-scale private Ethereum network with up to 32 nodes. Each HDSC contains the code of applying the Kaccak-256 hash function on a string for 1000 times; for simplicity, no input is required. The variables in the evaluation include: (i) N: the total number of full nodes; N varies from 8 to 32, with 32 as the default value. (ii) n: the number of executors for each HDSC; n varies from 6 to 12 with 6 as the default value. (iii) α: the required number of blocks that must have linearly extended a block before the block is considered stable; in this evaluation, we set $\alpha = n$. (iv) ρ: the number of HDSCs issued every minute, which varies from 1 to 4 with 2 as the default value.

We measure the following performance metrics: (i) $getResDelay$: the delay from when a HDSC gets ready till when the correct result is obtained. (ii) $finalizeDelay$: the delay from when a HSDC get ready till the HDSC has been completely processed.

The results are presented in Figure 5, where each point is the average of the measurements collected for one hour, and 95% confidence intervals are depicted as vertical lines.

As shown in Figure 5(a), when $N = 32$, $\rho = 2$ and $n = \alpha = 6$, the average delay from when a HDSC gets ready till when the execution result gets available is around 250 seconds, which can be explained as follows: Suppose a HDSC get ready at block x, its executors are the miners of blocks $x+1, x+2, \cdots, x+6$. Block $x+6$ gets stable only after blocks $x+7, \cdots, x+12$ have been added (due to $\alpha = 6$), which is the time when all the executors can start executing the HDSC code. After the execution finishes, commitments are posted to the following blocks, which get stable only after they are extended linearly by at least $\alpha = 6$ more blocks. Then, the executors can start revealing their results to the following blocks, which get stable also after being linearly extended by at least $\alpha = 6$ blocks. Hence, the results cannot be revealed before block $x + 24$. Because the block interval is usually between 10 and 20 seconds, the delay from when the HDSC gets ready till when the execution results are revealed is more than 200 seconds. After the results are revealed, 100 more seconds are further needed for the distribution of deposits to be performed and stabilized. This result indicates that, when the frequency of posting HDSCs to the blockchain is not high and each HDSC does not require a large number of executors, the delays are mainly due to: (i) the delays to wait for blocks get stable, (ii) the delay for the executors to be determined, and (iii) the several steps (i.e., result commitment step and the result revelation step) required by the proposed scheme. Here, factors (ii) and (iii) are the overheads introduced by the proposed scheme, and also the tradeoff for not involving all nodes in the execution.

Figure 5(a) also shows how the delays vary as n. As expected, when n increases (which also increase α), the delays increases accordingly. More over, as n increases, larger percentages (from 19% to 38%) of the nodes are demanded for each HDSC execution, which results in sharp increase of the delays. The phenomenon indicates that, when larger percentage of the nodes is required to execute a HDSC, not only that more computation power is wasted, but also that longer delays are incurred, as the increase in workload taken by each node decreases the system throughput. This is also



(a) Delays vs Number of Executors (b) Delays vs Number of Nodes (c) Delays vs HDSCs per minute

Figure 5: Delay vs Number of Executors, Nodes, and HDSCs per minute

echoed by Figure 5(b) which shows that, when $n = 6$, the larger is N (i.e., the smaller percentage of the network nodes is required to execute a HDSC), the shorter are the delays.

Figure 5(c) shows the change of delays with ρ, where 19% (i.e., 6 out of 32) of the nodes are demanded to execute each HDSC. As we can see, when ρ increases (i.e., more HDSCs are issued per minutes, the workload taken by each nodes also increases, which causes sharp increase of the delays. This further indicates the necessity of reducing the percentage of nodes required to execute each HDSC, especially when the number of HDSCs issued to the system gets large.

VI. Conclusions and Future Work

This paper proposed a game theory based, practical solution, to efficiently and securely execute heavy-duty smart contracts. Extensive game theoretic analysis was conducted to show the security and computational efficiency of the solution, even in face of collusion among the executors. The proposed solution was implemented atop Ethereum, and experimented in a small-scale private network, to demonstrate its practicality and compatibility with Ethereum. In the future, we plan to enhance the current implementation and conduct larger-scale experiments to further evaluate the solution.

Acknowledgement

This work is supported partly by NSF under grant CNS-1844591.

References

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

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

[3] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 51–68.

[4] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual International Cryptology Conference*. Springer, 2017, pp. 357–388.

[5] P. Daian, R. Pass, and E. Shi, "Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 23–41.

[6] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution," *arXiv preprint arXiv:1804.05141*, 2018.

[7] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti, "Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric," *arXiv preprint arXiv:1805.08541*, 2018.

[8] J. Lind, I. Eyal, F. Kelbert, O. Naor, P. Pietzuch, and E. G. Sirer, "Teechain: Scalable blockchain payments using trusted execution environments," *arXiv preprint arXiv:1707.05454*, 2017.

[9] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-ng: A scalable blockchain protocol," in *13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16)*, 2016, pp. 45–59.

[10] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 583–598.

[11] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 931–948.

[12] R. Pass and E. Shi, "Thunderella: Blockchains with optimistic instant confirmation," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 3–33.

[13] J. Eberhardt and S. Tai, "On or off the blockchain? insights on off-chaining computation and data," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2017, pp. 3–15.

[14] C. Li, B. Palanisamy, and R. Xu, "Scalable and privacy-preserving design of on/off-chain smart contracts," in *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 2019, pp. 7–12.

[15] C. Molina-Jimenez, I. Sfyrakis, E. Solaiman, I. Ng, M. W. Wong, A. Chun, and J. 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)*. IEEE, 2018, pp. 83–90.

[16] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1353–1370.

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

[18] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "Enforcing private data usage control with blockchain and attested off-chain contract execution," *arXiv preprint arXiv:1904.07275*, 2019.

[19] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Annual international cryptology conference*. Springer, 1991, pp. 129–140.

[20] P. Liu and W. Zhang, "A game theoretic approach for secure and efficient heavy-duty smart contracts," 2020. [Online]. Available: http://web.cs.iastate.edu/%7Ewzhang/hdsc.pdf