ROCK, PAPER, SCISSORS...Cheat — Verified Decentralized Game Play

Changping Chen, Ariel Hamlin, Jeffrey Lim, Manushaqe Muco

MIT

Version 1.0

May 13, 2015

1 Introduction

In our project we address the problem of how to implement a stateful multi-player game while minimizing trust in third-parties. By "stateful" we mean that there is some meaningful state that persists between separate interactions, as opposed to each interaction being self-contained, as in chess, tic-tac-toe, etc. A typical example of a stateful game is poker, since each hand affects the money balances held by each player, which persist between hands.

Most games that exist today prevent cheating either by centralization — requiring that all interactions be routed through a trusted third-party solely responsible for maintaining the game state; or else by obfuscating the software running on each player's device to prevent them from illegally altering the state.

In our project we seek a way to use cryptographic technology to alleviate these constraints. Our project implements a minimal game of this type as a proof-of-concept.

1.1 Overview

For this project we have implemented a minimal example of a game with these properties, which we call "Rock-Paper-Scissors-with-state" (RPSWS). The rules of the game are as follows:

1. Any player may initiate an encounter with another player, who may or may not choose to accept. (The two players are known as the *challenger* and the *defender* respectively.) No player may be in more than one encounter at the same time.

2. An encounter consists of a number of rounds. In each round, each player commits to their move (either Rock, Paper, or Scissors), and reveals the commitment to the other player. The winner of each round is found according to standard Rock-Paper-Scissors rules (Rock beats Scissors beats Paper beats Rock).

3. The player to win two out three rounds wins the encounter.

When the encounter concludes, each player's "skill rating" is adjusted according to a simple formula[1]. Players' skill ratings persist between encounters.

1.2 Architecture

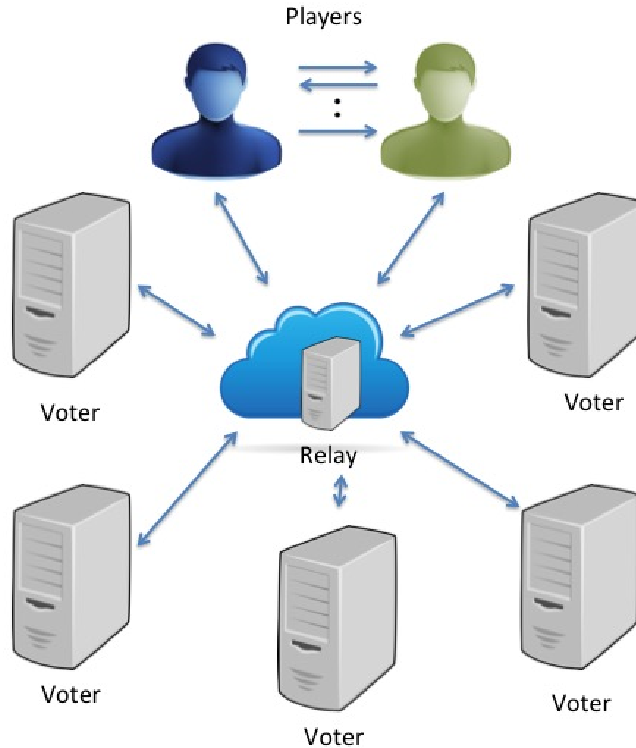

Figure 1: Overall Architecture

The game ecosystem as a whole may be conceptualized as a number of players who periodically interact with each other and with a "ledger" that represents the global state, available in common to all players.

The principle guiding our implementation is that each communication should involve as few parties as possible. Communication, when it does occur, goes over a peer-to-peer network for voters, and from point to point in the case of players. Players communicate with each other to play out encounters, which they then submit to the voters for verification and "disposition" (i.e. the process of updating the ledger to reflect the outcome of the encounter).

Accordingly, in our design we have taken pains to ensure that gameplay only requires the players to interact with the ledger at the beginning and end of an encounter. In particular,

[1]The winner gains (and the loser loses) an amount of points equal to $\left\lceil \frac{1000}{1+e^{\frac{W-L}{1000}}} \right\rceil$, where W, L are the skill ratings of the winner and loser before the encounter, respectively.

the players need not record their individual moves in the ledger as they make them; instead, the moves are assembled by the players themselves into a *encounter proof* transcript that they submit to the ledger at the end. Additionally, at the start, the players must query the ledger in order to record the beginning of their encounter; this prevents players from being in more than one encounter at the same time.

1.3 Threat Model

Our threat model is concerned with the fairness of the gameplay and of the maintenance of the game state. We consider threats arising from both malicious players and malicious voters.

Players may exhibit various malicious behaviors during the game encounter, of which we consider three types. First, a player might attempt to change their move upon seeing the opponents choice; this is undesirable in RPSWS or any other game in which players are supposed to move "simultaneously" without knowledge of their opponent's move. Second, a player may try to forge or replay in order to gain skill points without actually playing any games. Finally, a player may abort an encounter before a final outcome can be decided ("ragequitting").

The case where two players are both malicious and colluding is out of scope for our project; in any case, this scenario is unlikely to pose a threat in the context of an adversarial game.

We also consider the possibility of some amount of the voters being malicious and attempting to illegally alter the game state. They may attempt to alter account information or validate forged games in order to favor some players over others.

2 Related Work

2.1 Rock Paper Scissors Protocol Secure

An existing implementation by Thofmann [7], called Rock Paper Scissors Protocol Secure (RPSPS), is based on the same underlying premise as ours; namely, the use of a decentralized system to help support two-party games of RPS in which state is maintained between games. RPSPS uses Bitcoin as opposed to our proof-of-stake ledger system. The format of the transactions between players follows a similar pattern to ours, using commitments to ensure the integrity of moves.

However, despite these similarities, their adversarial model diverges significantly from ours. We consider malicious players in addition to their malicious distributed third-party (the "voters" in our system). Thus, they do not deal with the case of a malicious player replaying games or aborting mid-game. There is also no method for verifying the outcome of a game; the loser of an encounter has little incentive to actually hand over their stake.

2.2 Multiparty Computation (MPC) with a Malicious Majority

While the setting we are considering is very similar to that of MPC — two mutually distrusting parties who wish to compute a joint output — the security in our model is fundamentally

different. Of the main security properties of MPC, we are concerned primarily with the fairness of the outcome. We are not concerned with hiding the other player's input; in fact, we want players to be able to tell if they have won or not, so that they can verify the skill level changes determined by the outcome of the game. Even though we are only concerned with the fairness of the protocol, we can still apply results from MPC in the case where we have a malicious majority (in a two party interaction, even one malicious player is considered a majority) [2].

3 Player

3.1 Design

As mentioned in the previous section, transactions are submitted to the ledger by "players". A player is tied to an "account" (essentially, a public key) recorded in the ledger. While it is possible for a voter to be a player as well (in fact, this is how players can earn currency to fund their game transactions), our model does not require this.

We assume a supporting infrastructure for players to contact each other in order to initiate encounters. This involves account identity and network information (such as IP address or open port numbers). Players must be connected to each other during an encounter, except in the case of aborts, which we discuss in Section 5.3.

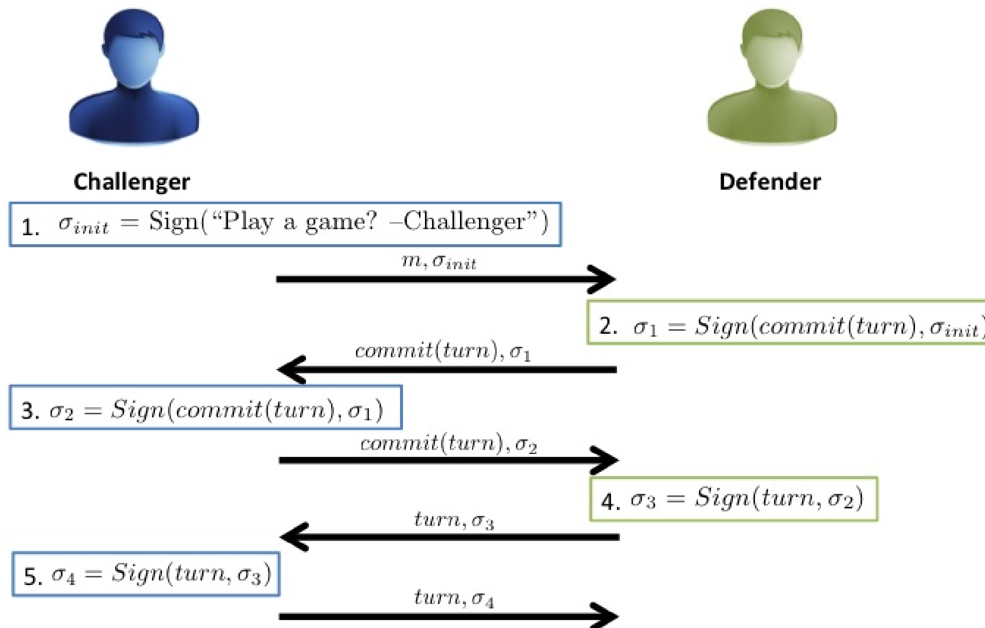

Figure 2: Player Interactions

Game play proceeds with a player initiating an encounter, as shown in Diagram 2 as Message 1. This initiator is called the *challenger* and the other the *defender*. It contains of the challenger's and defender's account IDs, and various encounter-specific values to help prevent replay and forging attacks, in a manner we will discuss later. This message is signed by the challenger. The defender may accept or reject the challenge.

If they accept, the gameplay advances with three rounds of RPS, each of which consists of four messages:

2. **Defender commitment:** The defender chooses a move (Rock, Paper, or Scissors) and calculates a commitment. In a signed message they send this commitment, *along with the signature of the previous message*. This prevents the message from being taken out of context and assembled into a fake encounter transcript later. At this time, the defender also submits an InitiateEncounter transaction to the voters, instructing them to mark the two players as currently being in an encounter with each other, including all of the values specified in Message 1.

3. **Challenger commitment:** Upon receives the defender's commitment, the challenger first queries the voters to verify that the ledger has been updated with the InitiateEncounter transaction. (This avoids the possibility of having an unregistered encounter that will not be allowed to alter the ledger.) Then, the challenger forms a commitment to their own move, again including the signature of the previous message.

4. **Defender reveal:** The defender responds by revealing their own move.

5. **Challenger reveal:** The challenger reveals their move.

This proceeds for three rounds. Afterward, the defender sends a "resolve" message to the challenger, and posts a CloseEncounter transaction to the voters, consisting of all of the messages sent so far. Once the voters have verified the validity of this transaction, they update the ledger accordingly.

3.2 Security Argument

We allow for the following limited malicious activity from players:

- Malicious parties (either the players themselves, or a third party who has access to the transcript) replaying a game transcript to the voters

- Either player quitting mid-encounter to prevent the game from reaching a conclusion

- Either player attempting to change their move after seeing the other's move

- Man-in-the-middle attacks between the two players

- A malicious party forging an encounter (or parts thereof) to post to the voters

Due to the fact that each player signs each transaction in player-to-player communication, as long as the user can validate the correct verification key belongs to the right user, a computationally bounded Eve cannot launch man-in-the-middle attacks. To do so would violate the EUCMA (existential unforgeability under chosen message attack) property of the signature scheme that we are using.

We use commitments to prevent retroactive changing of moves (which would otherwise allow the second mover to always win). However, these commitments must be *hiding*, in order to prevent the challenger, who receives the defender's commitment before sending their own, from extracting the value and committing to a winning move.

The most difficult malicious behavior between players to address (outside of abort) is preventing games from being forged. This is done by linking each message with the previous one, through the inclusion of the signature.

As a strawman construction, since we want unforgeability, we could simply sign the signature of the previous message to link the two. However, we cannot rely on this to prevent forgery, because of the small message space (3^4) it is easy for a player to build an exhaustive list of possible transactions between themselves and another player. Once this is built, they can piece together arbitrary transcripts that will appear authentic. Since we cannot rely on the entropy of the moves to make each message unique, we must add an additional source of entropy to the encounter.

Our solution is to include some notion of the current ledger state at the start of the encounter to the initial message — included in the "game-specific information" alluded to earlier. Each message that makes up part of this game transcript is now unique. We also do not have to worry about a malicious player assembling the exhaustive list for that particular point in the transcript because they are limited, at any given point, to only one game.

Denial of service, beyond the limited case of abort, is out of our scope of considered threats. This includes a challenger spamming the defender with game requests or somehow otherwise prevent a player from engaging with other players or the voters.

3.3 Future Work

Succinct Encounter Verification Our current protocol for encounter verification amongst the voters requires linear computation in the length of the encounter. Because the length of the encounter in RPSWS is fairly short, this is not difficult for the voters to do during the process of updating the ledger. However, for more complex games, the verification process will also be much more complex, potentially straining the computational resources of the voters, particularly if many players are all playing at the same time.

In order to solve this issues, we can turn to recent developments in Succinct Non-interactive Arguments of Knowledge (SNARKs). Work done by Ben-Sasson et al. [1] and Parno et al. [5] provide solutions in which the verification time of the proof of computation is sublinear in the size of the computation. In addition to verification time, they also offer an expressive set of computations for which it is possible to build verification proofs. In future work, when more complex games are developed, SNARKs may provide the answer to reducing the verification time to practical levels.

P2P Architecture In our model, players are assumed to have access to a third-party which provides routing information for players in order to initiate games. This is unnecessarily limiting for users to discover opponents and is very close to the third-party system this architecture was attempting to avoid. As a possible mitigation, it may be possible to apply concepts from peer-to-peer systems for routing and resource discovery.

Gateway Attacks If the player cannot make choices as to which voters they are willing to talk to, they are susceptible to *gateway attacks*. In our current architecture, communication between a player and voters takes place through a relay, and as such, players have no control of which voters they communicate with. As a result, their communications may be relayed to a malicious voter who may drop or otherwise mutate their transactions. Player choice of which voter they wish to communicate with would be an effective extension of our current architecture to mitigate gateway attacks. Then, a player can have some notion of "trust" in a particular voter and only communicate with those deemed trustworthy.

Player Censure There are several ways a player can detect malicious activity by their opponent: they try and reveal a move they did not commit to, their signature does not verify, or they never issued the InitiateEncounter transaction. In this case the player has no formal way to penalizing this behavior beyond choosing not to play another game with that player. In fact, if they choose to discontinue the game upon detecting malicious behavior, they themselves may be penalized for aborting the game prematurely. As a result, for future work, we propose extending functionality of player-to-player interaction to handle player-side censure of malicious behavior. There are several open problems in this case, as this faces issues similar to those raised in the fairness or attribution of abort — it may be difficult for the voters to determine which party is being malicious.

4 Voters

The ledger is implemented as a distributed network of *voters*, each of which maintains its own copy of the current ledger. Clients submit transactions to the voters through the relay, which then broadcasts them throughout the network; the voters update their ledgers based on these transactions.

The voters' responsibility is primarily to check transactions for *external validity* (i.e. consistency with the current state), as opposed to *internal validity* (i.e. the well-formedness of the transaction object itself), which can easily be checked by anyone without reference to the current state. However, voters do verify the internal validity of transactions in our design. A transaction that is internally valid will not necessarily be externally valid — for example, an InitiateEncounter transaction, even if validly signed, should be rejected if the two players are already in an encounter. Thus, even a small disagreement about the current state will lead to different transactions being accepted or rejected by different voters, which will lead to still further disagreement, and so on. It is therefore necessary that all voters come to consensus about which transactions to accept or reject — maintaining this consensus is more important than any particular ordering of transactions.

More precisely, this problem is known as *distributed Byzantine consensus*, referring to the fact that the system must withstand some amount of malicious behavior on the part of the voters. The most well-known system of this kind is Bitcoin [4], in which voters "vote" with their computing power by contributing to a proof-of-work chain. However, for our application we chose not to use Bitcoin (or a similar system), for two reasons: first, Bitcoin transactions take on the order of 10 minutes to confirm, which is too slow to accommodate reasonable gameplay; and second, proof-of-work is expensive for the voters in terms of hardware and electricity costs, which would be prohibitive for an application such as this that is primarily a form of entertainment rather than a financial instrument.

Instead, we propose a *proof-of-stake system*, where voters' votes are weighted according to how much "stake" they have – stake being a currency that can be transferred and used to pay for transaction fees).

The mechanism we propose is based on the Ripple consensus protocol [6, 3], which introduced the idea of the "continuously closing ledger". In Ripple, voting proceeds in stages, at the start of the consensus round, voters verify internally-valid transactions that they believe should be included to the ledger. Once the internal process is completed, transactions are broadcast to some subset of voters on the network. At each stage, each transaction must be approved by a certain fraction of the voters in order to advance to the next stage. This threshold increases in each stage, starting at 50% and increasing to 60%, 70%, and finally 80%, at which point the transactions are considered finalized, the ledger is updated, and the process restarts. Any internally-valid transactions that did not make it into one consensus round are deferred till the next; each round typically takes 4-5 seconds, which is fast enough to be a realistic mechanism for gameplay.

In contrast to our system, Ripple voters weights other voters' votes by membership in a *unique node list* (UNL) maintained by each voter — if a vote comes from someone in the UNL, it counts for 1; otherwise it counts for 0. The advantage of this is that it is not necessary to trust everyone in the UNL; it is only necessary to trust them not to collude with each other, which is a much less strict requirement. Nevertheless, the necessity to manually configure one's UNL is a disadvantage, which we aim to alleviate by proof-of-stake.

If the stake is distributed in such a manner that it is unlikely for a majority of the stake to be controlled by colluding entities, then it is reasonable to presume that it will remain so distributed even when individuals transfer their stake to others. If the stake can be distributed initially in this way (such as by giving 1/1000th of the stake to 1000 unrelated individuals), then by induction we can be confident that it will continue to be hard for an attacker to collect 51% of the stake. Stake is moved between individuals as part of transaction fees, or to add new voters into the voting pool.

It should be noted, however, that such economic arguments are quite imprecise and non-rigorous, as far as security is concerned. Research is still ongoing into the question of under what conditions Bitcoin, Ripple, and similar systems can be proved secure.

5 Mitigating Player-Voter Communication Attacks

In this section we discuss the mitigations for multiple games, replay, and abort attacks. It is worth noting that delay attacks in which game transactions are delayed in their posting

to the ledger are a special subset of abort attacks.

5.1 Preventing replay attacks

A *replay attack* is a form of attack in which a valid data transmission is maliciously repeated. This is carried out either by the originator of the message or by an adversary who intercepts the data and retransmits it. In our scenario, a valid game transaction is rebroadcast to the voters, by either the original players of the particular game, or by another individual who intercepts the transaction.

When a defender (Alice) receives a game request from the challenger (Bob), if he accepts, he will send a commitment and initialize the encounter with the voters. The voters have access to the most recent account states. An account state keeps track of the following information to prevent replay attacks: time the encounter begins at, time the encounter is supposed to end at, and the player the person is in encounter with. If the game is initialized with the voters, the account state is set to show that the player is in a game with the opponent, it started at a particular point, and finally, the encounter should end by a point agreed to by both players.

Let us assume that Alice wins. Based on the game transaction, the voters can calculate who won, but before they update the skill ratings of Alice and Bob they first must verify that the transaction is *externally valid* and not a replay attack. Consider the case where Alice wins - because she won, she may want to replay the game transaction again in order to gain more skill. There are two cases of replay that we need to handle:

1. After the game with Bob is over, Alice sends the same game transaction to the voters. In this case, the voters will compute that Alice won against Bob, the encounter began at time x, and that the current ledger value is y. However, the state for Alice will indicate that she is not in a game with Bob. In non replay attack, the state would show she is in a game with Bob, since the state is updated only after the voters compute the results. Because of this inconsistency, the voters will conclude that this is a replay attack.

2. Alice acts smart about it, and she starts a new game with Bob. However, while the game is still going she replays the old game transaction. In this case, Alice's state will show that she is currently in an encounter against Bob. Alice is trying to get around the state check by claiming to have won the current game by replaying an old game transaction. However, since she started a new game with Bob, her account information was set for the encounter to start at a specific time, for example x. Upon receiving the fraudulent game transaction, the voters can obtain the start ledger of the encounter and determine that is occurred *before* the current encounter started. At this point the voters can conclude this is a replay attack and not include the transaction in the consensus.

5.2 Multiple games at once

We say that a player is trying to play multiple games at once, in the case when the player is trying to play another game while he is in the middle of playing a game, or has committed

to playing one.

Let us assume that Alice is playing a game with Bob, or has committed to play a game with Bob. Alice's current account state will reflect this fact – listing her as in encounter with Bob and that it started at a certain time. If Alice wants to initiate another game, the voters will check her current state and see that she is already committed to playing against Bob. A player is not allowed to play multiple games at once, so the voters will not include the initiate game request for Alice and the game will be discontinued.

5.3 Abort

As alluded to previous sections, we incentivize proper behavior in the case of one player prematurely ending, or aborting, the game. Ishai et al [2], show that in the case of a malicious majority, it is possible for the parties in the MPC protocol to know who aborts. Conversely, parties, such as the voters, who do not take place in this computation, cannot ascertain who the aborting party is. Within these results, we attempt to motivate 'proper' behavior within reason.

Our protocol allows a user who believes the game has been aborted to post this fact to the ledger. The player includes the proof of the game so far in this notification which is stored in the users' account state. The opposing player now has a set amount of time to refute the abort claim. At this point, there are two possibilities:

1. The accused player rebuts the abort claim and posts their own move to the ledger to continue the game,
2. Or the accused player fails to post and the skill involved comes out in favor of the accuser

Due to the fact that each transaction with the ledger has a set cost in stake, it is far cheaper to continue game play through player-to-player interaction — not to mention more timely. In gameplay through the ledger, players must wait for the ledger to be resolved before they can make the next move. It is worth noting, that while the voters do not know which player truly aborted, the players themselves do know. That being said, the honest player can blacklist the malicious player for future games.

6 Implementation

We created three individual components as part of our software prototype: relay, voter, and client. For more information about the implementation and the code refer to Section 7.1

Relay instantiates a TCP server that voters connect to. By keeping a list of connected voters, it can relay a message received from any voter to all other voters to simulate an idealized lossless peer-to-peer network. In the current implementation, a client does not directly join the voter network, but instead posts and receives message from voters through a separate HTTP server interface on the relay.

Voter is a game proof verifier that records and maintains the global game state in a SQLite database. A voter upon receiving game proof from clients through relay's HTTP interface,

will validate the proof according to predefined rules and broadcast its validation result via relay to other voters. A voter also receives validation results from other voters. Once a consensus is reached, the SQLite database representing the ledger is updated accordingly. A voter must have a valid account already existing on the ledger to do so.

All components take advantage of asynchronous network programming featured in Python Tornado library. It abstracts low-level TCP networking API in a event-based paradigm and uses advanced Python coroutine feature to support concurrent access. This greatly eases the implementation of communications between components.

We use PyCrypto for the cryptographic primitives, including RSA encryption and digital signatures. For hash functions, we use Python's built-in `hashlib` module.

Since no implementation for a commitment scheme was readily available, we created our simple commitment scheme. In our design, a user concatenates the value to be committed with a 20-byte randomly generated padding, and computes the SHA256 hash of the concatenated string as the commitment. To reveal the commitment, a user sends its committed value and previously generated padding to the verifier who computes the correct commitment and checks if it matches the previously received commitment.

We represent users in our system using their unique public-keys. In addition, similar to the Bitcoin network, we use a 48-character long account ID generated from the trailing 36 bytes of the hash of a user's public key encoded in DER format. This significantly reduces the length of an account ID, while preserving the one-to-one mapping between users and account IDs. However, when users sign a message, they must include their public key along with the signature since the verifier may not necessarily have the signer's public key for verification. The verifier must also check if the signer's account ID corresponds to the received public key.

7 Conclusion

We have designed and implemented Rock-Paper-Scissor-with-State to provide a verifiable, decentralized, game platform. Our voter network leverages existing consensus methods to allow for decentralized global state and our players provide verifiable game encounters. It is our hope that this general platform with its security guarantees and protocol flexibility can provide a base for more complex game protocols that include verifiable randomness, increased number of parties, and more efficient proof verification.

7.1 Code Repository

Links to our implementation and information about usage can be found at the following locations:

- Rock-Paper-Scissors-Cheat Github Repository
- Code usage

References

[1] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. Cryptology ePrint Archive, Report 2013/507, 2013. http://eprint.iacr.org/.

[2] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. Cryptology ePrint Archive, Report 2015/325, 2015. `http://eprint.iacr.org/`.

[3] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. 2015.

[4] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system,? http://bitcoin.org/bitcoin.pdf.

[5] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 238–252, Washington, DC, USA, 2013. IEEE Computer Society.

[6] David Shwartz, Noah Youngs, and Arthur Britto. The ripple protocol consensus algorithm. 2014.

[7] Thofmann. Rock paper scissors protocol secure. https://github.com/thofmann/rock-paper-scissors-protocol-secure, 2015.