SmartCast: An Incentive Compatible Consensus Protocol Using Smart Contracts

Abhiram Kothapalli
kothapa2@illinois.edu

Andrew Miller
soc1024@illinois.edu

Nikita Borisov
nikita@illinois.edu

Abstract

Motivated by the desire for high-throughput public databases (i.e., "blockchains"), we design incentive compatible protocols that run "off-chain", but rely on an existing cryptocurrency to implement a reward and/or punishment mechanism. Our protocols are incentive compatible in the sense that behaving honestly is a weak Nash equilibrium, even in spite of potentially malicious behavior from a small fraction of the participants (i.e., the BAR model from Clement et al. [8]). To show the feasibility of our approach, we build a prototype implementation, called SmartCast, comprising an Ethereum smart contract, and an off-chain consensus protocol based on Dolev-Strong [11]. SmartCast also includes a "marketplace" smart contract that randomly assigns workers to protocol instances. We evaluate the communication costs of our system, as well as the "gas" transaction costs that are involved in running the Ethereum smart contract.

Keywords— atomic broadcast, TRB, game theory, Ethereum, smart contracts

1 Introduction

Bitcoin and related cryptocurrencies have sparked renewed interest in decentralized consensus protocols, as exemplified by the so-called blockchain technologies. It turns out that many applications (including complementary currencies, certificate revocation [15, 7], directory authorities for p2p networks like Tor [10]), benefit from a globally agreed-upon sequence of transactions. Currencies such as Bitcoin and Ethereum use the proof-of-work mining to distribute the responsibility for maintaining the blockchain integrity to a large collection of parties; the integration of mining with a financial reward makes this collection difficult to subvert. However, the global nature of this ledger creates some inherent costs, both in terms of transaction costs and the agreement latency. An alternative approach is what has been termed a *permissioned ledger*, where the role of miners is taken by a trusted coalition of parties, whose motivation to properly follow the protocol is assumed to come externally.

Several applications of blockchains, however, would benefit from a middle ground between these two extremes. Loosely defined coalitions, such as food banks, cooperatives, or student organizations, are some times in need of a blockchain-like ledger for tracking membership benefits or exchanges between sister organizations; however, they would not have the resources to directly operate a reliable collection of "miners," nor, necessarily agree on a set of trusted parties. At the same time, directly using cryptocurrency for account deposits might limit their accounting flexibility and incur non-trivial transaction costs.

Our approach creates a system where workers who act to enforce integrity are financially rewarded for their correct participation in the process, as monitored by other workers and enforced through an Ethereum smart contract. Our protocol draws inspiration from a consensus protocol designed by Clement et al. [8], where honest participation is shown to be a *rational* strategy for participants trying to maximize their utility. Their protocol, however, assumes that workers derive *intrinsic* utility from the correct functioning of the protocol and requires an infinite time horizon; in our scenario, which we believe to be more realistic, we expect consensus to be enforced by inherently disinterested parties whose only motivation is financial. This *extrinsic* reward dramatically simplifies the protocol design and improves its efficiency. Our protocol requires only occasional communication with the Ethereum blockchain through the smart contract, thus minimizing transaction costs.

To design our protocol, we create a *generic* transform that renders any existing protocol where communication is the dominant cost *incentive-compatible*. We show that, under this transform, honest participation is a weak Nash equilibrium in a worst-case utility model, which was previously used by Clement et al. [8].

To show the feasiblity of our approach, we build a prototype implementation, called SmartCast, comprising an Ethereum smart contract and an "off-chain" consensus protocol (based on the Dolev-Strong [11] broadcast protocol). . We evaluate the communication costs of our system, as well as the "gas" transaction costs that are involved in running the Ethereum smart contract. We additionally describe how these protocols can be deployed in practice with random consensus nodes.

1.1 Related Works

Several previous works have proposed using cryptocurrencies to enforce properties in off-chain protocols. Bentov and Kumaresan's protocol [1] guarantees either a fair output or else financial compensation to each honest party, but requires significant collateral deposits. In contrast, our weak Nash equilbirium notion guarantees that parties cannot benefit by deviating. In a separate line of work, Garay et al. design a general framework to build protocols that are resilient against rational adversaries [12]. We instead design a protocol transformer that can achieve resilience for a certain class of protocols. To the best of our knowledge, we are the first to harness smart contracts for the purpose of Byzantine fault tolerance.

2 Background and Preliminaries

2.1 Network Model

Our basic computation model is the standard point-to-point network setting with synchronous authenticated channels. We consider a fixed set of parties $\mathcal{N}$, where an individual party is denoted $p \in \mathcal{N}$. The protocol proceeds in rounds of communication, with the exact order of messages in each round may be arbitrary (i.e., chosen by the adversary). Messages not delivered within the round are invalid. Each party is associated with a common-knowledge public signing key to send authenticated messages. Our model accounts for Byzantine failures. The adversary can choose to corrupt a subset of nodes $\mathcal{B} \subset \mathcal{N}$, giving them complete control over these nodes. $|\mathcal{B}|$ is bounded by a parameter b.

2.2 Smart Contract Protocols

Public cryptocurrencies [5] (or "blockchain") systems, such as Bitcoin [18] and Ethereum [21], provide a decentralized platform for programmable money. These can be used as general-purpose trusted third parties, but with caveats. For instance, they can be trusted for correctness, but do not provide any inherent privacy. For some applications, privacy can be provided by a layer of multi-party computations and zero-knowledge proofs [13, 2]. A second caveat is that blockchain transactions are expensive (because they are fully replicated throughout the entire cryptocurrency network), so it typically is not cost-effective to carry out protocols directly on top of the blockchain.

A protocol in the smart contract model is therefore most effective with two components: 1) A smart contract program, which receives reports from nodes about each other, and dispenses rewards at the end; and 2) Local code for each of the parties to execute, including interactions with the smart contract and participation in "off-chain" subprotocols. We also assume a rushing adversary, who can observe the smart contract transactions sent by non-Byzantine parties before submitting transactions on behalf of the Byzantine parties.

2.3 Utilities in the BAR Model.

We adapt the The Byzantine-Altruistic-Rational (BAR) fault tolerance model from Clement et al. [8] to the smart contract setting. The BAR model is a game-theoretic layer on top of the standard distributed protocol execution model. That is, we view the choice of what code to run (i.e., either following the protocol or deviating in some arbitrary way) as a strategic decision.

We associate each "off-chain" protocol message with a *cost* to the sender of that message, determined by the total size of the messages sent. However, we ignore any other costs of computation, storage, and other resources. We thus assume that the total utility of each party therefore depends on the monetary payments disbursed by the smart contract, minus the cost of the messages they send. Since the protocol execution is probabilistic, unless indicated otherwise we are concerned with the expected utility.

As Clement identifies, in an ordinary protocol (i.e., without the smart contract incentive mechanism in place), parties may be able to profit by deviating from the protocol — in particular by withholding messages to reduce their costs (i.e., by acting "lazy"). Thus the high level approach is to punish lazy nodes.

A strategy profile $\vec{\sigma}$ defines a program for each party p in $\mathcal{N}$ to run. Given a protocol ρ, we use the symbol $\vec{\rho}$ to denote the *prescribed strategy*, in which every party follows the protocol correctly.

While standard distributed systems models feature a worst-case adversary, and standard game models feature a set of strategic (i.e., "rational") players, the intersection of these has yet to be studied widely. Clement proposes the following notion of "worst-case utility," which we also adopt.

Definition 1. *Worst-case Utility. The worst case utility $\bar{u}_p(\sigma)$ for a rational player $p \in \mathcal{N}$ is where p follows strategy σ, every non-Byzantine player in $N - \mathcal{B} - \{p\}$ follows the prescribed strategy, $\vec{\rho}_{N-\mathcal{B}}$, and the choice of Byzantine players $\mathcal{B}$ and their strategies $\vec{\tau}_{\mathcal{B}} \in \mathcal{S}_{\mathcal{B}}$ are chosen to minimize the resulting utility u_p. This is defined more precisely as:*

$$\bar{u}_p(\sigma) \triangleq \min_{\mathcal{B} \subset \mathcal{N} : |\mathcal{B}| \leq b} \circ \min_{\vec{\tau} \in S_{\mathcal{B}}} \circ\ u_p(\vec{\rho}_{\mathcal{N}-\mathcal{B}-\{p\}} + \sigma + \vec{\tau}_{\mathcal{B}}) \tag{1}$$

Our goal is then to show that the prescribed strategy is a worst-case *weak Nash equilibrium*, i.e., $\bar{u}_p(\rho_p) \geq \bar{u}_p(\sigma)$ for any σ. That is, a rational party cannot *improve* their expected utility by following any other deviant protocol σ. This solution concept could be thought of as modeling paranoid players who think that other parties (up to b of them colluding) are "out to get them."

2.4 Synchronous Byzantine Agreement

Alternative definitions of consensus primitives abound in the distributed systems literature. Perhaps the strongest of these — and the one most naturally suited to our application scenario — is "atomic broadcast." This primitive allows any of the N protocols parties to submit input values, and the parties all reach agreement on an ordered sequence of values that at least includes the inputs from each honest party. Atomic broadcast could thus be described as the "blockchain" primitive in today's post-Bitcoin parlance.

Below we provide a more formal definition of this primitive, adapted for the synchronous setting. We assume that each input value is bounded by a maximum message size C, and that the protocol finally terminates after a maximum number of rounds $r^\dagger$.

Definition 2. *Bounded Atomic Broadcast: Given a set of players $\mathcal{N}$, each process p in $\mathcal{N}$ receives inputs $m_{p,r} \in \{0,1\}^C$ in round r.*

- *(Termination):* after a bounded number of rounds $r^\dagger$, each correct process terminates.
- *(Agreement):* The sequence of outputs $V_{p,r}$ in round r by each correct process p are all identical, i.e. $\forall r, \forall p, q \in \mathcal{N} - \mathcal{B}. V_{p,r} = V_{q,r}$.
- *(Validity):* every input from a correct node (received before $r \leq r^\dagger$) is included in $V_{r^\dagger}$.

Looking ahead, in Section 3.4 we implement an atomic broadcast protocol by composing a simpler primitive, called Terminating Reliable Broadcast (TRB). In TRB, one of the parties is designated as the leader, and only the leader may input messages. Thus in TRB there is no need to apply an ordering to messages from different sources, and if the leader is faulty then the parties may need to output a default value $\perp$.

Definition 3. *Terminating Reliable Broadcast Given a set of players $\mathcal{N}$, among which one, D, is designated the leader and receives an input $m \in \{0,1\}^C$ (i.e., within some bounded message size of C bits), a Terminating Reliable Broadcast protocol must satisfy the following properties:*

- *(Termination):* Every correct process p delivers some value $m \in \{0,1\}^C \cup \{\perp\}$ after a bounded time r^*.
- *(Agreement):* If any correct process delivers m, then all correct processes deliver m.
- *(Validity):* If the leader D is correct, then every correct process delivers D's input m.

Alternative network models. Although our SmartCast protocol relies on a synchronous network model. This is generally considered a strong assumption. Other protocols such as PBFT [6] provide security in the more challenging weakly synchronous setting — they meet the lower bound in this model, which is $b \leq N/3$. However, synchrony is an assumption we must take anyway if we rely on a smart contract system in the style of Bitcoin and Ethereum. It is not clear how to adapt our protocol to the asynchronous setting, since we would not be able to detect whether a message was omitted by a party or just delayed in the network.

3 Smart Contracts for Incentive Compatible Protocols

In this section we present our main contribution, a protocol transformer, $\mathsf{SmartBAR}(\cdot)$, which transforms an arbitrary synchronous protocol with costly communication, π into an incentive compatible protocol $\mathsf{SmartBAR}(\pi)$. As an application, in Section 3.4 we apply this transformation to yield an incentive-compatible consensus protocol, called SmartCast.

At a high level, $\mathsf{SmartBAR}(\cdot)$ adds a smart contract layer to π that implements a reward/punishment mechanism. In an ordinary protocol (i.e., without this incentive mechanism in place), parties may be able to profit by deviating from the protocol — in particular by withholding messages to reduce their costs (i.e., by acting "lazy"). To ensure that laziness is not profitable, our protocol enlists the honest parties to detect lazy nodes and the smart contract to punish them.

The transformation works for an arbitrary synchronous protocol π that satisfy the following assumptions. First, each correct party in π terminates after a bounded number of rounds r^*, for some parameter r^*. Second, the total number of bits between any pair of parties is bounded by a value M. We call a protocol that satisfies these an $(r^*, M)-$*bounded* synchronous protocol.

Since the transformation runs π in place, any fault tolerance properties of π still carry over to $\mathsf{SmartBAR}(\pi)$. In particular, if π tolerates b faults, and we prove that running is a b-worst-case equilibrium, then the security of the overall protocol security reduces to the assumption of strategic behavior among the rational remaining parties.

3.1 The Protocol Transformer $\mathsf{SmartBAR}(\cdot)$

The transformed protocol $\mathsf{SmartBAR}(\pi)$ runs π with the following minimal modifications:

- Modification 1: We impose a predictable communication pattern so that nodes can detect if another is cutting costs by not forwarding messages. Our predictable communication pattern requires that in each protocol instance, node p must send every node q the maximum possible total message size M. If fewer than M message bits are sent by the end of the protocol, then dummy messages are sent to make up the difference.
- Modification 2: We impose a penalty on nodes that fail to forward messages, by implementing the following rules:
  - Each node keeps track of the total message bits received from each other node.
  - At the end of the protocol, if fewer than M bits have been received by p from q, then p sends a report $R_{p,q} = \texttt{enemy}$ to the smart contract. Otherwise, if at least M bits have been received, then p sends a report $R_{p,q} = \texttt{friend}$.
  - The smart contract waits until the final round of the protocol r^* to collect status reports from all nodes $p \in \mathcal{N}$. Finally, the smart contract determines the payout to each party by deducting a penalty of θ (a parameter discussed shortly) for each `enemy` report about that party.

Alternative definitions of `enemy` Note that we propose a relatively lenient definition for `enemy` as a node that does not send at least M bits. This protects honest nodes with harmless or negligible deviations from being marked as dishonest by other honest nodes. On the other hand, we can follow a much stricter definition of enemy by marking nodes that do not send at least M bits, send incorrect bits, send more than M bits, and so on. This leads to additional protocol security by barring more forms of misbehavior, but may

Protocol $\mathsf{Smart}(\pi)$ for a bounded synchronous protocol π, a set of parties $\mathcal{N}$, and a maximum number of Byzantine nodes $b < |\mathcal{N}| - 2$.

Let r^* be a bound on the final round before π terminates.

Let M bound the total size of messages sent between any pair of parties in π.

Local program (for node p).

- Run the given protocol π_p.
- For each received message m, parse m as either an ordinary message $\texttt{PASS}(m')$ (in which case pass m' through to π_p) or else a padding message $\texttt{DUMMY}(0^*)$, in which case discard this message.
- For each outgoing message m generated by π_p, intended for player q, send $\texttt{PASS}(m)$ to q.
- At the final round r^*, let $M_{p,q}$ be the total size of messages sent so far to q (including any messages sent during this round). If $M_{p,q} < M$, then send a padding message $\texttt{DUMMY}(0^{M-M_{p,q}})$.
- After r^*, for each player $q \neq p$, let $M_{q,p}$ be the total size of messages received from q. If $M_{q,p} < M$, then set $R_{p,q} := 0$ (an enemy report). Otherwise, set $R_{p,q} := \theta$ (a friend report). Finally, send a transaction containing $\texttt{report}(\vec{R_p})$ to the smart contract, where $\vec{R_p} = \{R_{p,q}\}_{q \neq p}$ is the vector of all of the reports from p about each other player q.

Smart contract program.

- The contract is parameterized by a set of players $\mathcal{N}$, identified by their addresses (i.e., public keys).
- The contract must be initialized with an endowment (a quantity of digital currency) of at least $E \geq (|\mathcal{N}|)(|\mathcal{N}| - 1) \cdot \theta$, where $\theta = \frac{|\mathcal{N}|-1}{|\mathcal{N}|-1-b} M$.
- The contract contains an entry point $\texttt{report}(\vec{R_p})$, which when invoked by party p, stores the argument vector $\vec{R_p}$.
- By a fixed deadline time T, the contract receives a report $R_{p,q} \in [0, \theta]$ from each party $p \in \mathcal{N}$ about each other party $q \in \mathcal{N}$. Any reports that are not received in time are treated as a default value of 0.
- After time T, for each $p \in N$,
  - determine the reward as the sum of reports about p, so $\mathsf{reward}_p := \sum_{q \in \mathcal{N} | q \neq p} R_{q,p}$, and send reward_p to player p

Figure 1: Our protocol transformer, $\mathsf{Smart}(\cdot)$, which provides incentive-compatibility for an arbitrary synchronous protocol. Each party pads outgoing messages to the maximum size, and reports to the smart contract about any “lazy” peers.

unnecessarily penalize honest nodes that perform harmless or negligible deviations.

The entire SmartBAR$(\cdot)$ protocol is defined in Figure 1. For simplicity, we assume the smart contract is initialized with an endowment $E \geq N(N-1)\cdot\theta$. In practice, this endowment might be provided by collecting collateral deposits from the participants or collecting usage fees from users of the system, as described shortly in Section 3.5. We next describe how the parameter θ is determined in order to satisfy the worst-case equilibrium notion.

3.2 Rationality Analysis

We now prove that following the SmartBAR protocol is a worst-case weak Nash equilibrium. The utility for party $p \in N$ as a function of a strategy vector $\vec{\sigma}$ is $u_p(\vec{\sigma}) = \mathit{benefits}_p(\vec{\sigma}) - \mathit{costs}_p(\vec{\sigma})$. The overall benefits will be decided by reward_p and the overall cost is $\mathit{cost}_{msg} + \mathit{cost}_{report}$. In the following, we use the notation $\vec{\sigma}_{N-\{p\}} + \vec{\sigma}_p$ to denote the union of the strategy vectors $\vec{\sigma}_{N-\{p\}} + \vec{\sigma}_p$.

In order to prove that rational parties gain the highest utility by following the recommended protocol, we take the following steps: First we show a lower bound that following the protocol earns p a minimum utility u^*, regardless of the adversary's choice of strategies. Next, we partition the space of alternative strategies into classes based on how they behave towards honest nodes. We define a simple family of strategies, called the "indiscriminate" strategies, which act as representatives of these classes. We can prove that the indiscriminate strategies perform just as well (in the worst-case) as any other strategy. Finally, we show how to choose the protocol parameter θ so that u^* is an upper bound for the utility of any indiscriminate strategy. The setting of θ directly determines the overall collateral cost (i.e., the required endowment) for the protocol.

Lemma 1. *Regardless of the strategy $\vec{\tau}_{\mathcal{B}}$ followed by Byzantine parties, if p follows $\vec{\rho}_p$, then p obtains at least* $\bar{u}_p(\rho) \geq u^*$ *where* $u^* \triangleq (N-1)\theta - (N-1)M - b\theta$.

Proof. The ideal reward of the protocol is initially set to be $(N-1)\theta$. The prescribed strategy sends all possible messages, incurring the maximum message cost $(N-1)M$. Since all the non-Byzantine nodes report p as a `friend`, the maximum report cost can be at most $b\theta$, which occurs when all b Byzantine nodes report `enemy`. □

This bound holds regardless of how the protocol parameter θ is chosen. This worst-case utility is incurred when the Byzantine parties follow the spiteful strategy.

The indiscriminate strategies, α_γ. We next turn towards proving an upper bound on the utility of deviating from the prescribed strategy ρ. We first define a family of simple strategies, α, which we call the indiscriminate strategies. Looking ahead, these strategies will serve as representatives for a partioning of the overal strategy space. The indiscriminate strategies α by a fraction $0 \leq \gamma \leq 1$, such that α_γ misbehaves towards each other node with probability γ. More precisely, α_γ is defined as follows: At the beginning of the game, for each other party q a coin is flipped with probability γ (for some arbitrary percentage γ). If the coin flip succeeds, then p refuses to send any messages to q; otherwise p sends messages to q according to the ordinary protocol.

The strategy α_γ clearly causes p to incur a message cost of $(1-\gamma)(N-1)M$. Since this strategy witholds messages from exactly $\gamma(N-1-b)$ honest uncorrupted parties in expectation, the worst-case expected report cost is $(b + \gamma(N-1-b))\theta$. We therefore have the following claim:

Claim 1. *The worst-case expected utility for the strategy α_γ is*

$$\bar{u}_p(\alpha_\gamma) = (N-1)\theta - (N-1)(1-\gamma)M - (b + \gamma(N-1-b))\theta \tag{2}$$

The Spiteful Strategy, δ. Following Clement et al. [8], we define a particular adversarial strategy, called the spiteful strategy, which serves as a worst-case adversary (as we will see shortly). The spiteful strategy initially behaves according to the prescribed strategy, but in the final round it always reports `enemy` for player p, causing p to be punished.

If rational party p could determine which nodes were corrupted, then p would be able to cut his losses by withholding messages from just the nodes in $\mathcal{B}$. The spiteful strategy, however, blends in with the honest parties. As shown by the following lemma, this means p can do no better than to withhold messages from other nodes chosen uniformly at random, as with the indiscriminate strategy α_γ. In the following, we say that player p follows an *acceptable message sequence* towards player q if p sends q a total of at least M bits.

Lemma 2. *Consider a strategy σ_γ, such that in an execution with all honest parties (i.e., with the strategy vector $\{\sigma_\gamma\} + \rho_{\mathcal{N}-\{p\}}$), party p sends an unacceptable message sequence to exactly $\gamma(N-1)$ nodes in expectation. Then the worst-case utility $\bar{u}(\sigma_\gamma)$ is at most $\bar{u}(\alpha_\gamma)$.*

Proof. Let γ_q be the probability that σ_γ sends an unacceptable message sequence to party $q \in \mathcal{N} - \{p\}$ when all parties besides p follow the protocol. By assumption, we know that

$$\sum_{q \in \mathcal{N}-\{p\}} \gamma_q = \gamma.$$

First, note that against the spiteful adversary, p incurs an expected message cost of at least $(1-\gamma)(N-1)M$. Next, to bound the report cost, we will choose $\mathcal{B} \subseteq \mathcal{N} - \{p\}$, with $|\mathcal{B}| = b$, such that we minimize $\sum_{q \in \mathcal{B}} \gamma_q$. This minimization guarantees that p sends an unacceptable message sequence to at least $(N-1-b)\gamma$ honest nodes in expectation, resulting in an expected report cost of at least $(b + (N-1-b)(\gamma))\theta$. □

Note the above proof above holds regardless of whether probabilities γ_q are independent.

Choosing the parameter θ. We want to choose θ so that deviating from ρ cannot improve the worst-case expected utility. Starting from Lemma 2, it will suffice if we can guarantee that $\bar{u}_p(\rho) \geq \bar{u}_p(\alpha_\gamma)$ for all γ. We therefore solve the following:

$$\bar{u}_p(\rho) \geq \bar{u}_p(\alpha_\gamma) \tag{3}$$

$$(N-1)M + b\theta \leq (N-1)(1-\gamma)M + (b + (N-1-b)(\gamma))\theta \tag{4}$$

$$\frac{N-1}{N-1-b}M \leq \theta \tag{5}$$

Theorem 1. *If π is a synchronous protocol that terminates after r^* rounds and each party sends a maximum of M message bits to each other party, then the transformed protocol $\mathsf{Smart}(\pi)$ is a worst-case weak Nash equilibrium.*

Proof. When the SmartTRB protocol is instantiated with θ defined as in Equation 5, from Lemma 1 we have that the worst-case expected utility when p follows the protocol $\bar{u}_p(\rho)$ is at least as good as any indiscriminate strategy $\bar{u}_p(\alpha_\gamma)$. And from Lemma 2, we know that the indiscriminate strategies perform as well in the worst-case as any other strategies. □

3.3 Comparison with the BAR Primer [8].

Our protocol and analytical framework is adapted from the bar model of Clement et al. [8], but with several significant differences.

While Clement's model requires an infinitely repeated game, our model considers the bounded case. In the infinite settings, rational parties simply play tit-for-tat, immediately and irrevocably "retaliating" against nodes that misbehave, preventing them from all future rewards. In a finite setting, a node could misbehave in the final round without fear of retaliation.

Additionally Clement's model assumes that nodes gain a positive utility from the correction execution of the protocol itself. Alternatively, in our model, nodes gain a positive utility monetary payments disbursed by the smart contract. We believe our utility model is more realistic, especially in a marketplace setting (like that discussed in Section 3.5) where the participants in a protocol instance are randomly assigned from some population of available workers.

Together, these two modelling differences require a significant change to the protocol and proof. First, while "retaliation" in Clement's model involves requiring nodes to send expensive "penance" messages (since that is the only way to inflict a punishment in their model), the smart contract provides a direct alternative. Second, in the finite setting we must rule out deviant strategies that withhold messages in a possibly randomized way, even in the last round, as though "guessing" at which parties might be corrupted. We overcome this by introducing a new family of "indiscriminate strategies" that serve as simple representatives of the full strategy space. Finally, like Clement, our proof involves a "spiteful strategy," that acts as a worst case adversary. However, the "spiteful strategy" is different in our model: it misbehaves only in the final round, after it is too late for the victim p to retaliate.

3.4 SmartCast: An Incentive Compatible Consensus Protocol

As an application of our general protocol transformation, we now describe how to apply our $\mathsf{SmartBAR}(\cdot)$ transformation to a classic synchronous protocol, $\mathsf{DolevStrong}$ [11], in order to obtain an incentive-compatible off-chain consensus protocol.

The Dolev-Strong protocol for Terminating Reliable Broadcast. The Dolev-Strong protocol is a classic algorithm for synchronous byzantine agreement using signatures, that achieves optimal resilience by tolerating $N-1$. However, it provides no explicit incentives for participants to follow. As seen in Clements et al., individual participants in the protocol can reduce their computational cost by omitting messages.

The protocol runs for $b+1$ rounds, where the leader D sends a signed message containing its input to each of the other nodes in the first round. Each node that receives the leader's message in the first round "accepts" the message, and then appends their own signature and relays the message to every other node. If the leader fails to send a message to some node p, some other node q will relay the message to p in any round r, as long as the relay contains at least r signatures. p will then continue to relay the message. If the leader equivocates, it is possible that a node accepts two or more distinct values. In this case, a node outputs $\perp$, and only relays the first two such values received. In total, each node must therefore send a maximum of $2N$ total messages, each containing an input value and up to $b+1$ signatures.

We provide a listing of the Dolev-Strong algorithm in Figure 2, adapted from Kumaresan's thesis [14]. For a proof of security we refer the reader to [11, 14].

We let $D \in \mathcal{N}$ denote a designated leader. We let $m \in \{0,1\}^C$ denote the sender D's input, and sk_D its secret key.

- (Stage 1): The leader D sends $(m, \mathsf{sign}_{\mathsf{sk}_D}(m))$ to every party. It then outputs m and terminates the protocol. Each other party p initializes $\mathsf{ACC}_p := \emptyset$, and $\mathsf{SET}_p := (v \mapsto \emptyset)$, a mapping from values to (initially empty) sets of signatures.
- (Stage 2): In rounds $r = 1, ..., b + 1$, perform the following:
  - If a pair (v, SET) is received from some q, with $v \in \{0,1\}^C$, and if SET contains valid signatures on v from at least r distinct parties including the leader D, and if ACC_p contains only 0 or 1 values, then p updates $\mathsf{ACC}_p := \mathsf{ACC}_p \cup \{v\}$, and $\mathsf{SET}_p[v] := \mathsf{SET}_p[v] \cup \mathsf{SET}$.
  - Each party p checks whether any value $v \in \{0,1\}^C$ was newly added to the set of accepted values ACC_p during round $r - 1$. In this case, p computes $\mathsf{sign}_{\mathsf{sk}_p}(v)$, and sends $(v, \mathsf{SET}_p[v] \cup \{\mathsf{sign}_{\mathsf{sk}_p}(v)\}$ to every other party.
- (Stage 3): If $\mathsf{ACC}_p = \{v\}$ for some v, then p outputs v. Otherwise p outputs $\perp$.

Figure 2: Definition of the DolevStrong protocol [11] for Terminating Reliable Broadcast (adapted from Kumaresan [14])

From Reliable Broadcast to Atomic Broadcast. Atomic broadcast further guarantees that messages can be committed by any node, not just a leader. In a synchronous network, atomic broadcast can be easily built from terminating reliable broadcast, simply by having nodes take turns becoming leaders. In brief, each node maintains a buffer of input values that have not been committed yet. When it is node p's turn as leader, p broadcasts the set of elements in its buffer. When each turn ends, the nodes remove any newly committed elements from their buffers.

3.5 Deploying Consensus Protocols with Smart Contracts

So far, we have discussed protocols assuming a fixed set of parties, with collateral provided abstractly by a benefactor. We now describe an alternative deployment scenario where many independent SmartCast instances are run concurrently, and where the participants in each are randomly drawn from a large population of potential workers. Our idea is to build a smart contract-based marketplace, SmartCast-Market, that matches up workers to protocol instances.

A naïve approach might be to allow participants to join a SmartCast instance a first-come-first-serve basis. This naïve solution would be vulnerable to Sybil attacks, where malicious nodes join as fast as they can with numerous pseudonyms, hoping to fill up all of the slots in a protocol and therefore crowd out honest nodes. Instead, our solution is to allow nodes to join a pool of workers, and to allow task creators to deposit collateral and add to a pool of pending tasks. Every epoch, workers are assigned to tasks in a randomized batch. This prevents nodes from gaining too much influence within any particular protocol instance.

If all participants in an instance follow the protocol, then the total payment for a task must be $P = N(N-1)\theta$. In principle, this could be collected from a combination of upfront payment from the task creator, along with collateral deposits from the participants themselves. Since participation is voluntary, we should ensure as a guideline that workers never lose money by participating in the protocol. Thus if they deposit collateral, they must

```
contract SmartCast {
    mapping(address => uint) playermap;
    bool[] reported;
    address[] players;
    uint[] rewards;
    uint theta;
    uint deadline; // Deadline to receive reports

    function assert(bool b) internal { if (!b) throw; }
    modifier after_ (uint T) { if (block.number >= T) _; else throw; }
    modifier before(uint T) { if (block.number <  T) _; else throw; }
    modifier onlyplayer() { if (playermap[msg.sender] != 0) _; else throw; }

    function SmartCast(address[] _players, uint _theta, uint _deadline) {
        var N = players.length;
        // Each player earns up to N * theta if they receive all good reports
        assert(msg.value == N * N * _theta);
        theta = _theta;
        deadline = _deadline;
        for (var p = 0; p < _players.length; p++) {
            players.push(_players[p]);
            rewards.push(0);
            playermap[_players[p]] = (p+1);
        }
    }
    function report(uint[] reports) onlyplayer before(deadline) {
        var p = playermap[msg.sender] - 1;
        assert(!reported[p]); reported[p] = true; // only report once
        assert(penalties.length == players.length);
        for (var q = 0; q < reports.length; q++) {
            var report = reports[q];
            assert(report >= 0);
            assert(report <= theta);
            rewards[q] += report;
        }
    }
    function withdraw() onlyplayer after_(deadline) {
        var i = playermap[msg.sender] - 1;
        if (!msg.sender.send(balance[i])) throw;
        balance[i] = 0;
    }
}
```

Figure 3: Implementation of the Smart contract in Solidity.

get at least that collateral back (in expectation) despite a worst-case adversary. However, since the parameter $\theta = \frac{N-1}{N-1-b}M$ is chosen minimally, in the worst-case each honest party just breaks even, receiving only $(N-1-b)\theta$ in payment but incurring an equal message cost of $(N-1)M$. Therefore there is no opportunity for collateral deposits to contribute to the necessary endowment. Thus the task creator must pay up-front the maximum payment $N(N-1)\theta$, although the maximum total message cost is only $N(N-1)M$. Hence, the task creator potentially pays an overhead of $\frac{N-1}{N-1-b}$ above the raw cost of the resources used.

4 Implementation and Evaluation

To evaluate the practical limitations of the SmartCast protocol, we develop a prototype implementation of both the Dolev-Strong consensus algorithm and an Ethereum smart contract capable of assigning various nodes to arbitrary consensus tasks.

Our Dolev-Strong implementation is written in Python, using ordinary threads and TCP sockets, with messages signed using the `ed25519` signature scheme. We evaluated our protocol by running on a network of up to 16 Amazon EC2 instances. To simulate realistic

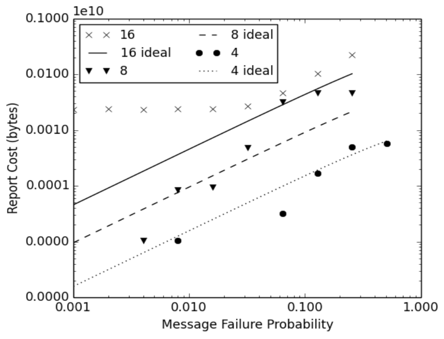

Figure 4: Penalties imposed on nodes vs. message failure probability.

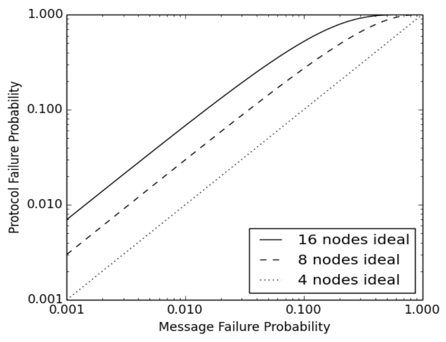

Figure 5: Consistency failure vs. message failure probability (analytic only).

network delays, we used the Linux traffic control tool to limit bandwidth to 5mbps and impose a 100ms latency per message.

In the synchronous network model, messages between honest parties are simply guaranteed to be delivered within a given time bound. However, in reality, it is necessary to choose this timeout parameter concretely, based on estimates of network performance and on a tolerance for failure. Too short a timeout, and messages between otherwise-honest parties may fail to be delivered in time. In our experiments the payload for each broadcast is a constant size of 1 megabyte (i.e., the size of a Bitcoin block today). We benchmarked the network and computation load by performing several trial computations and measuring the resulting message delivery time, and then fitting a normal distribution to the results.

We first analyze the effects of message failure on the individual participants bottom line. If a node p fails to deliver a message to q in time, then q will inflict a punishment on p. Since each node is required to send 2 messages, if each message fails with probability ζ, we expect the expected cost of punishments to be $(N-1)(1-(1-\zeta)^2)$. In Figure 4 we compare the actual punishment incurred in our experiment with this expected line.

Message delivery failures can also lead to inconsistent outputs. In the worst case, if the maximum number of b nodes are actively attacking the network, then even a single failed message among the remaining nodes can lead to inconsistency. This occurs in the following scenario: suppose b nodes (including the leader) are corrupted, and send no messages at all until round b (the next-to-last round). At the beginning of round b, one of the corrupted nodes sends a message to a single honest node p containing a value v and b signatures. The node p will accept (and output) the value v, and relay it to the remaining $N-1-b$ honest nodes. If even one of these nodes fails to receive this final-round message, then it will output an inconsistent value $\perp$. Thus given b malicious nodes, and assuming messages fail independently with probability ζ, the uncorrupted nodes could output inconsistent values with probability $1-(1-N^2)$ probability (these are plotted in Figure 5).

4.1 Ethereum Smart Contract

We implemented the smart contract component of SmartCast in Ethereum's Solidity programming language. Our implementation includes:

- A smart contract for collecting reports, and handling payments. The entire program listing is shown in Figure 3.

- A smart contract implementing the "Marketplace" described in Section 3.5.
- A test framework using `pyethereum`, allowing us to measure the "gas costs" (i.e., transaction fees) for varying numbers of parties.

The Solidity language syntax resembles Javascript, and the intended effect of each line should be clear in context (though we imagine readers may be skeptical of the details, given several recent high-profile failures caused by subtle Solidity quirks [17, 9, 16]). Fortunately, the Smart Contract program listing in 3 fairly closely matches the pseudocode in Figure 1. We explain a few Solidity idioms that readers are likely to be unfamiliar with. Solidity supports "modifier" macros, which are convenient for specifying preconditions which must hold before a function is called (or else they `throw` an error). Furthermore, although the pseudocode disburses all rewards immediately upon the deadline, Ethereum does not directly support time-triggered events, thus the indirect `withdraw` function is necessary.

The Marketplace Contract. We also implemented a Solidity version of the "marketplace" smart contract described in Section 3.5. Below we describe its high level functions. For space, we omit the full Solidity code listing; the full code will be made available online.

- `registerTask`: creates a new task, configured with any application-specific parameters (e.g., description of a validation condition or a list of approved clients). The task creator must include payment sufficient to pay the workers for the task.
- `registerWorker`: allows a worker to sign up, depositing any necessary collateral.
- `finalize`: shuffles the list of workers and list of tasks, and then assigns workers to tasks until either a) no tasks are remaining, or b) not enough workers are available to fill the remaining task. For each fully-assigned task, spawn a new instance of the SmartCast contract. Return any deposited collateral to workers who were not assigned to a task, and refund payment to task creators whose tasks were not fulfilled.

Our protocol relies on a random beacon; our prototype simply uses `block.blockhash(0)` as a source of randomness, although this is known to be manipulable by miners [4, 19].

Ethereum Benchmarks. We tested our smart contract implementation using the `pyethereum.tester` framework. Table 1 shows the required gas costs for varying configurations of our application. We show results for only a few possible configurations: we increase the number of parties P, but always fill two tasks with two workers left over. The `finalize` method is the most expensive, since it grows with $O(N)$ when shuffling the list of workers. However, the `registerWorker` and `registerTask` methods are each invoked N times, and thus contribute about equally to the total.

Ethereum imposes a per-block (and hence, per-transaction) gas limit, which miners can vote to change gradually over time. Although the simulator easily supports these large transactions, today's Ethereum blockchain enforces a limit of approximately 2 million gas units, which the `finalize` operation busts when $P \geq 20$ (as underlined in Table 1). To avoid this limit, an alternative approach would be to spread the `finalize` operation over several contract invocations. This would require more complicated code, since each invocation would need to explicitly load and save its internal state. Our application provides a motivation for higher-level programming abstractions for transactions spanning multiple blocks.

Table 1: Smart contract gas costs (normalized to dollars, based on current Ethereum parameters and price (as of Nov 14 2016)). Underlined costs are infeasible, exceeding Ethereum's current per-block gas limit.

(N,P,T)	`registerWorker`		`registerTask`		`finalize`		Tot	
	Gas	(USD)	Gas	(USD)	Gas	(USD)	Gas	(USD)
$(4, 10, 2)$	110743	2.7¢	153347	3.8¢	1215702	30.4¢	2614826	65.4¢
$(8, 18, 2)$	110743	2.7¢	153347	3.8¢	1863111	46.6¢	4234665	\$1.05
$(16, 34, 2)$	110743	2.7¢	153347	3.8¢	<u>2966784</u>	74.0¢	6678740	\$1.70
$(32, 66, 2)$	110743	2.7¢	153347	3.8¢	<u>5271727</u>	\$1.32	12047459	\$3.01

Alternative implementation in Bitcoin. Our SmartBar protocol could still function using only Bitcoin's multi-signature transactions. The parties and the benefactor would generate N^2 transactions, where each transaction $T_{p,q}$ rewards θ to party q conditionally on a signature from p.

5 Conclusion and Future Work

We have adapted the work of Clement et al. [8] to the "smart contract" world, using cryptocurrencies to provide incentive compatibility for off-chain consensus protocols. Though we give a specific instantiation based on the Dolev-Strong protocol for reliable broadcast, our protocol is expressed as a generic transformation for arbitrary synchronous protocols.

Although the incentive compatibility notion we have adapted from Clement et al. [8] is described as "worst-case," modeling arbitrary Byzantine failures, many plausible attacks yet lie outside this model. In particular, our definition counterintuitively rules out "bribery" attacks, which are well-known though have not been observed in practice [3, 20]. Notice that the "worst-case" notion is from the point of view of an individual participant; since accepting a bribe makes an individual party richer, this is excluded by definition. Additionally, our utility model assumes unilateral deviation, which rules out collusion attacks. Incorporating both bribery and collusion into our model remains an important open problem.

References

[1] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *International Cryptology Conference*, pages 421–439. Springer, 2014.

[2] Iddo Bentov and Ranjit Kumaresan. How to Use Bitcoin to Design Fair Protocols. In *CRYPTO*, 2014.

[3] Joseph Bonneau. Why buy when you can rent? bribery attacks on bitcoin consensus. *Bitcoin Research Workshop*, 2016.

[4] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On bitcoin as a public randomness source. Cryptology ePrint Archive, Report 2015/1015, 2015. `http://eprint.iacr.org/2015/1015`.

[5] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. IEEE Symposium on Security and Privacy, 2015.

[6] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[7] Melissa Chase and Sarah Meiklejohn. Transparency overlays and applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 168–179. ACM, 2016.

[8] Allen Clement, Harry Li, Jeff Napper, Jean-Philippe Martin, Lorenzo Alvisi, and Mike Dahlin. Bar primer. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 287–296. IEEE, 2008.

[9] Kevin Delmolino, Mitchell Arnett, Ahmed E Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. *Bitcoin Research Workshop*, 2016.

[10] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[11] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[12] Juan Garay, Jonathan Katz, Ueli Maurer, Bjoern Tackmann, and Vassilis Zikas. Rational protocol design: Cryptography against incentive-driven adversaries. Cryptology ePrint Archive, Report 2013/496, 2013. `http://eprint.iacr.org/2013/496`.

[13] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 839–858, May 2016.

[14] Ranjit Kumaresan. Broadcast and verifiable secret sharing: New security models and round optimal constructions. 2012.

[15] Ben Laurie, Adam Langley, and E Kasper. Certificate transparency. *Network Working Group Internet-Draft, v12, work in progress. http://tools. ietf. org/html/draft-laurie-pki-sunlight-12*, 2013.

[16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[17] David Z. Morris. Blockchain-based venture capital fund hacked for $60 million. `http://fortune.com/2016/06/18/blockchain-vc-fund-hacked/`, June 2016.

[18] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `http://bitcoin.org/bitcoin.pdf`, 2008.

[19] Cecile Pierrot and Benjamin Wesolowski. Malleability of the blockchain's entropy. Cryptology ePrint Archive, Report 2016/370, 2016. `http://eprint.iacr.org/2016/370`.

[20] Jason Teutsch, Sanjay Jain, and Prateek Saxena. When cryptocurrencies mine their own business. *Bitcoin Research Workshop*, 2016.

[21] Gavin Wood. Ethereum: A secure decentralized transaction ledger. `http://gavwood.com/paper.pdf`, 2014.