Arbitrum: Scalable, private smart contracts

Harry Kalodner
Princeton University

Steven Goldfeder
Princeton University

Xiaoqi Chen
Princeton University

S. Matthew Weinberg
Princeton University

Edward W. Felten
Princeton University

Abstract

We present Arbitrum, a cryptocurrency system that supports smart contracts without the scalability and privacy limitations of previous systems such as Ethereum. Arbitrum, like Ethereum, allows parties to create smart contracts by using code to specify the behavior of a virtual machine (VM) that implements the contract's functionality. Arbitrum uses mechanism design to incentivize parties to agree off-chain on what a VM would do, so that the Arbitrum miners need only verify digital signatures to confirm that parties have agreed on a VM's behavior. In the event that the parties cannot reach unanimous agreement off-chain, Arbitrum still allows honest parties to advance the VM state on-chain. If a party tries to lie about a VM's behavior, the verifier (or miners) will identify and penalize the dishonest party by using a highly-efficient challenge-based protocol that exploits features of the Arbitrum virtual machine architecture. Moving the verification of VMs' behavior off-chain in this way provides dramatic improvements in scalability and privacy. We describe Arbitrum's protocol and virtual machine architecture, and we present a working prototype implementation.

1 Introduction

The combination of *digital currencies* and *smart contracts* is a natural marriage. Cryptocurrencies allow parties to transfer digital currency directly, relying on distributed protocols, cryptography, and incentives to enforce basic rules. Smart contracts allow parties to create virtual trusted third parties that will behave according to arbitrary agreed-upon rules, allowing the creation of complex multi-way protocols with very low counterparty risk. By running smart contracts on top of a cryptocurrency, one can encode monetary conditions and penalties inside the contract, and these will be enforced by the underlying consensus mechanism.

Ethereum [31] was the first cryptocurrency to support Turing-complete stateful smart contracts, but it suffers from limits on scalability and privacy. Ethereum requires every miner to emulate every step of execution of every contract, which is expensive and severely limits scalability. It also requires the code and data of every contract to be public, absent some type of privacy overlay feature which would impose costs of its own.

1.1 Arbitrum

We present the design and implementation of Arbitrum, a new approach to smart contracts which addresses these shortcomings. Arbitrum contracts are very cheap for verifiers to manage. (As explained below, we use the term *verifiers* generically to refer to the underlying consensus mechanism. For example, in the Bitcoin protocol, Bitcoin miners are the verifiers.) If parties behave according to incentives, Arbitrum verifiers need only verify a few digital signatures for each contract. Even if parties behave counter to their incentives, Arbitrum verifiers can efficiently adjudicate disputes about contract behavior without needing to examine the execution of more than one instruction by the contract. Arbitrum also allows contracts to execute privately, publishing only (saltable) hashes of contract states.

In Arbitrum, parties can implement a smart contract as a *Virtual Machine (VM)* that encodes the rules of a contract. The creator of a VM designates a set of *managers* for the VM. The Arbitrum protocol provides an *any-trust* guarantee: any one honest manager can force the VM to behave according to the VM's code. The parties that are interested in the VM's outcome can themselves serve as managers or appoint someone they trust to manage the VM on their behalf. For many contracts, the natural set of managers will be quite small in practice.

Relying on managers, rather than requiring every verifier to emulate every VM's execution, allows a VM's managers to advance the VM's state at a much lower cost

to the verifiers. Verifiers track only the hash of the VM's state, rather than the full state. Arbitrum creates incentives for the managers to agree out-of-band on what the VM will do. Any state change that is endorsed by all of the managers (and does not overspend the VM's funds) will be accepted by the verifiers. If, contrary to incentives, two managers disagree about what the VM will do, the verifiers employ a bisection protocol to narrow the disagreement down to the execution of a single instruction, and then one manager submits a simple proof of that one-instruction execution which the verifiers can check very efficiently. The manager who was wrong pays a substantial financial penalty to the verifiers, which serves to deter disagreements.

Parties can send messages and currency to a VM, and a VM can itself send messages and currency to other VMs or other parties. VMs may take actions based on the messages they receive. The Verifier tracks the hash of the VM's inbox.

The architecture of the Arbitrum VM and protocol are designed to make the task of resolving disputes as fast and simple for the verifiers as possible. Details of the design appear later in the paper.

Arbitrum dramatically reduces the cost of smart contracts. If participants behave according to their incentives, then verifiers will never have to emulate or verify the behavior of any VM. The only responsibility of verifiers in this case is to do simple bookkeeping to track the currency holdings, the hash of a message inbox, and a single hashed state value for each VM. If a participant behaves irrationally, it may require the verifiers to do a modest amount of extra work, but the verifiers will be (over-)compensated for this work at the expense of the irrational party.

As a corollary of the previous principle, Arbitrum VMs can be private, in the sense that a VM can be created and execute to completion without revealing the VM's code or its execution except for the content and timing of the messages and payments it sends, and (saltable) hashes of its state. Any manager of a VM will necessarily have the ability to reveal information about that VM, but if managers want to maintain a VM's privacy they can do so.

Arbitrum is consensus-agnostic, meaning that it assumes the existence of a consensus mechanism that publishes transactions, but the Arbitrum design works equally well with any consensus mechanism, including a single centralized publisher, a quorum-based consensus system, or Nakamoto consensus as used in Bitcoin [26]. Additionally, an existing smart contract system can serve as this consensus mechanism assuming it can encode Arbitrum's rules as a smart contract. In this paper, we refer to the consensus entity or system as the *Verifier* (and the participants in the said consensus system as the *verifiers*).

1.2 Structure of the paper

The remainder of the paper is structured as follows. In section 2 we discuss the difficulties of implementing smart contracts efficiently, and we present the *Participation Dilemma*, a new theoretical result on participation games showing that one approach to incentivize smart contract verification may not work. In section 3 we describe Arbitrum's approach, and in section 4 we provide more details of Arbitrum's protocol and virtual machine architecture, which together allow much more efficient and privacy-friendly verification of the operations of virtual machines implementing smart contracts. Section 5 describes our implementation of Arbitrum and provides some benchmarks of performance and the sizes of proofs and blockchain transactions. Section 6 surveys related work, and section 7 concludes the paper.

2 Why Scaling Smart Contracts is Difficult

Supporting smart contracts in a general and efficient way is a difficult problem. In this section we survey the drawbacks of some existing approaches.

2.1 The Verifier's Dilemma

The most obvious way to implement smart contract VMs is to have every miner in a cryptocurrency system emulate every step of execution of every VM. This has the advantage of simplicity, but it imposes severe limits on scalability.

The high cost of verifying VM execution may manifest as the *Verifier's Dilemma* [22]. Because transactions involving code execution by a VM are expensive to verify, a party that is supposed to verify these transactions has an incentive to free-ride by accepting the transactions without verifying them, in the hope that either (1) misbehavior is deterred by other parties' doing verification, or (2) any discrepancies will not be detected by other potential verifiers because they also do not perform verification. This can lead to an equilibrium in which some transactions are accepted with little or no verification. Conversely, in a scenario in which all miners are honestly doing the verification, a miner can exploit this by including a time-consuming computation that will take the other miners a significant amount of time to verify. While all of the other miners are doing the verification, the miner that included this computationally heavy transaction can get a head-start on mining the next block, giving it a disproportionate chance of collecting the next block reward. This dilemma exists because of the high cost of verifying VM execution.

2.2 The Participation Dilemma

One approach to scaling verification (as used in, *e.g.*, TrueBit [30]) relies on participation games, a mechanism design approach that aims to induce a limited but sufficient number of parties to verify each VM's execution. These systems face what we call the *Participation Dilemma*, of how to prevent Sybil attacks in which a single verifier, who may or may not be honest, claims to be multiple verifiers, and in doing so can drive other verifiers out of the system.

2.3 Participation Games

In this section we prove new formal barriers to approaches based on *participation games*. The idea is that players will "participate" in a costly process. Consider the following game:

- There are n players, who may pay 1 to participate.
- Participating player i chooses a number of Sybils $s_i \geq 1$. Non-participating players set $s_i = 0$.
- Player i receives reward $s_i \cdot f(\sum_j s_j)$, where $f : \mathbb{N} \to \mathbb{R}_+$ is a reward function.

In the context of this paper, think of participating as "verifying a computation." It costs something to verify the computation, but once you've verified it, you can claim to have verified it from any number of additional Sybils for free, and these Sybils are indistinguishable from "real" verifiers. The goal would then be to design a participation game (i.e. a reward function $f(\cdot)$) such that *in equilibrium*, no player has any incentive to Sybil, and a desired number of players participate, so that the apparent number of verifiers equals the actual number of separate players who were verifiers.

The authors of TrueBit correctly observe that the family of functions $f_c(m) = c \cdot 2^{-m}$ make great candidates for participation games. Specifically, for any target k of participating players, the participation game with reward function $f(m) = (2^k + 0.5) \cdot 2^{-m}$ has a unique (up to symmetry) pure Nash equilibrium where every player has $s_i \in \{0, 1\}$, and exactly k players participate. In fact, an even stronger property holds: it is always a best response for any player to set $s_i \leq 1$![1] We call such reward functions *One-Shot Sybil-Proof* (formal definition in Appendix A). This initially makes participation games seem like a promising avenue for verifiable smart contracts, as One-Shot Sybil-proof reward functions exist.

However, a problem that prior work fails to resolve is that smart contract verification is a *repeated game*. In repeated games, there are numerous other equilibria that don't project onto Nash equilibria of their one-shot variants. For intuition, recall the classic prisoner's dilemma:[2] if the game is only played once, then the unique Nash equilibrium is for both players to defect (and defecting is even a strictly dominant strategy). However, in the repeated prisoner's dilemma, there are numerous other equilibria including the famous Tit-For-Tat, and Grim Trigger strategies [29].

We discuss the formal model for repeated games (which is standard, but not the focus of this paper) in Appendix A. But the point is that repeated games allow for players to sacrifice the present in order to save for the future. For example, the following is an equilibrium of the repeated participation game with $f(m) = (4.5) \cdot 2^{-m}$. Player one uses the strategy: set $s_1 = 2$ in all rounds. Player $i > 2$ sets $s_i = 0$ in all rounds. Player 2 uses the strategy: if in either of the previous two rounds, $\sum_{j \neq 2} s_j \leq 1$, set $s_2 = 1$. Otherwise, set $s_2 = 0$.

Note that all players aside from player 1 are certainly best responding. They currently get utility zero (because player 1 sets $s_1 = 2$ every round, and they therefore all set $s_i = 0$). If they instead participated in any round, they would get negative utility. Player 1 on the other hand, is also best responding! This is because if they decreased their number of Sybils in any round, it would cause player 2 to participate in the next two rounds (formal proof in appendix).

Note that this equilibrium is not at all unnatural: players > 1 are simply reacting to what the market looked like in the previous rounds. Player 1 is staying one step ahead of the game and realizing that no matter what, there are going to be two participants in equilibrium, so player 1 might as well be all of them rather than share the reward. In fact, this is not a property specific to the reward function $c \cdot 2^i$, but *any reward function*.

Theorem 1. *Every One-Shot Sybil-Proof participation game admits a Nash equilibrium where only one player participates.*

In Appendix A, we provide a proof of Theorem 1, as well as a discussion of possible outside-the-box defenses. These defenses seem technically challenging (perhaps impossible) to implement, but we are are not claiming this provably. However, simulations do indicate that the cost to implement these defenses scales linearly with the computational power of a single player, which may render them impractical (if they are indeed even possible).

As a result, approaches based on this type of participation game, including those proposed in prior work [30, 32], appear to be unable to prevent Sybil attacks that undermine confidence in the verification of smart contracts.

[1] That is, no matter what the other players do, player i is strictly happier to set $s_i = 1$ than $s_i > 1$.

[2] There are two players. Both get payoff 1 if they both defect, and payoff 2 if they both cooperate. If one cooperates and the other defects, the defector gets 4 and the cooperator gets 0.

3 Arbitrum System Overview

In this section we give an overview of the design of Arbitrum.

3.1 Roles

There are four types of roles in the Arbitrum protocol and system.

The **Verifier** is the global entity or distributed protocol that verifies the validity of transactions and publishes accepted transactions. The Verifier might be a central entity or a distributed multiparty consensus system such as a distributed quorum system, a worldwide collection of miners as in the Nakamoto consensus protocol [26], or itself a smart contract on an existing cryptocurrency. Because the Arbitrum design is agnostic as to which type of consensus system is used, for brevity we use the singular term Verifier for whatever consensus system is operating.

A **key** is a participant in the protocol that can own currency and propose transactions. A key is identified by (the hash of) a public key. It can propose transactions by signing them with the corresponding private key.

A **VM (Virtual Machine)** is a virtual participant in the protocol. Every VM has code and data that define its behavior, according to the Arbitrum Virtual Machine (AVM) Specification,which is included in the extended version of this paper. Like keys, VMs can own currency and send and receive currency and messages. A VM is created by a special transaction type.

A **manager** of a VM is a party that monitors the progress of a particular VM and ensures the VM's correct behavior. When a VM is created, the transaction that creates the VM specifies a set of managers for the VM. A manager is identified by (the hash of) its public key.

3.2 Lifecycle of a VM

An Arbitrum VM is created using a special transaction, which specifies the initial state hash of the VM, a list of managers for the VM, and some parameters. As described below, the state hash represents a cryptographic commitment to the VM's state (i.e., its code and initial data). Any number of VMs can exist at the same time, typically with different managers.

Once a VM is created, managers can take action to cause that VM's state to change. The Arbitrum protocol provides an *any-trust* guarantee: any one honest manager can force the VM's state changes to be consistent with the VM's code and state, that is, to be a valid execution according to the AVM Specification.

An *assertion* states that if certain preconditions hold, the VM's state will change in a certain way. An assertion about a VM is said to be *eligible* if (1) the assertion's preconditions hold, (2) the VM is not in a halted state, and (3) the assertion does not spend more funds than the VM owns. The assertion contains the hash of the VM's new state and a set of actions taken by the VM, such as sending messages or currency.

Unanimous assertions are signed by all managers of that VM. If a unanimous assertion is eligible, it is immediately accepted by the Verifier as the new state of the VM.

Disputable assertions are signed by only a single manager, and that manager attaches a currency deposit to the assertion. If a disputable assertion is eligible, the assertion is published by the Verifier as pending. If a timeout period passes without any other manager challenging the pending assertion, the assertion is accepted by the Verifier and the asserter gets its deposit back. If another manager challenges the pending assertion, the challenger puts down a currency deposit, and the two managers engage in the *bisection protocol*, which determines which of them is lying. The liar will lose its deposit.

A VM continues to advance its state as described above, until the VM reaches a halted state. At this point no further state changes are possible, and the Verifier and managers can forget about the VM.

3.3 The Bisection Protocol

The bisection protocol begins when a manager has made a disputable assertion and another manager has challenged that assertion. Both managers will have put down a currency deposit.

At each step of the bisection protocol, the asserter bisects the assertion into two assertions, each involving half as many steps of computation by the VM, and the challenger chooses which half it would like to challenge. They continue this bisection protocol until an assertion about a single step (i.e., the execution of one instruction by the VM) is challenged, at which point the asserter must provide a one-step proof that the Verifier can check. The asserter wins if they provide a correct proof; otherwise the challenger wins. The winner gets their deposit back and also takes half of the loser's deposit. The other half of the loser's deposit goes to the Verifier.

The bisection protocol is carried out via a series of blockchain transactions made by the asserter and challenger. At each point in the protocol a party has a limited time interval to make their next move, and that party loses if they fail to make a valid move by the deadline. The Verifier only needs to check the facial validity of the moves, for example, checking that a bisection of an assertion into two half-sized assertions is valid in the sense that the two resulting assertions do indeed compose to yield the original assertion.

3.4 The Verifier's Role

Recall that the Verifier is the mechanism, which may be a distributed protocol with multiple participants, that verifies transactions and publishes verified transactions. In addition to storing a few parameters about each VM such as a list of its managers, the Verifier tracks three pieces of information about each VM that change over time: the hash of the VM's state, the amount of currency held by the VM, and the hash of the VM's inbox which holds messages sent to the VM. The state of a VM is advanced, corresponding to execution of the VM's program, by the Verifier's acceptance of assertions made by the VM's managers.

An assertion that is challenged cannot be accepted by the Verifier, even if the asserter wins the challenge game. Instead, an assertion is "orphaned" when it is challenged.[3] After the challenge game is over, the asserter has the option of resubmitting the same assertion, although this would obviously be foolish if the assertion is incorrect.

The protocol design ensures that a single honest manager can always prevent an incorrect assertion from being accepted, by challenging it. (If somebody else challenges the assertion before the honest manager can do so, the assertion is still prevented from being accepted, even if the challenger is malicious.) An honest manager can also ensure that the VM makes progress, by making disputable assertions, except that a malicious manager can delay progress for the duration of one bisection protocol at the cost of half of a deposit, by forcing a bisection protocol that it knows it will lose.

3.5 Key Assumptions and Tradeoffs

Arbitrum allows the party who creates a VM to specify that VM's code, initial data, and set of managers. The Verifier ensures that a VM cannot create currency but can only spend currency that was sent to it. Thus a party who does not know a VM's state or who does not like a VM's code, initial data, or set of managers can safely ignore that VM. It is assumed that parties will only pay attention to a VM if they agree that the VM was initialized correctly and they have some stake in its correct execution. Any party is free to create a VM that is obscure or unfair; and other parties are free to ignore it.

By Arbitrum's *any-trust* assumption, parties should only rely on the correct behavior of a VM if they trust at least one of the VM's managers. One way to have a manager you trust is to serve as a manager yourself. We also expect that a mature Arbitrum ecosystem would include manager-as-a-service businesses that have incentives to maintain a reputation for honesty, and may additionally accept legal liability for failure to carry out an honest manager's duties.

One key assumption that Arbitrum makes is that a manager will be able to send a challenge or response to the Verifier within the specified time window. In a blockchain setting, this means the ability get a transaction included in the blockchain within that time. While critical, this assumption is standard in cryptocurrencies, and risk can be mitigated by extending the challenge interval (which is a configurable parameter of each VM).

Two factors help to reduce the attractiveness of denial of service attacks against honest managers. First, if a DoS attacker cannot be certain of preventing an honest manager from submitting a challenge, but can only reduce the probability of a challenge to p, the risk of incurring a penalty may still be enough to deter a false assertion, especially if the deposit amount is increased. Second, because each manager is identified only by a public key, a manager can use replication to improve its availability, including the use of "undercover" replicas whose existence or location is not known to the attacker in advance.

Lastly, a motivated malicious manager can indefinitely stall a VM by continuously challenging all assertions about its behavior. The attacker will lose at least half of every deposit, and each such loss will delay the progress of the VM only for the time required to run the bisection protocol once. We assume that the creators of a VM will set the deposit amount for the VM to be large enough to deter this attack.

3.6 Benefits

Scalability. Perhaps the key feature of Arbitrum is its scalability. Managers can execute a machine indefinitely, paying only negligible transaction fees that are small and independent of the complexity of the code they are running. If participants follow incentives, all assertions should be unanimous and disputes should never occur, but even if a dispute does occur, the Verifier can efficiently resolve it at little cost to honest parties (but substantial cost to a dishonest party).

Privacy. Arbitrum's model is well-suited for private smart contracts. Absent a dispute, no internal state of a VM is revealed to the Verifier. Further, disputes should not occur if all parties execute the protocol according to their incentives. Even in the case of a dispute, the Verifier is only given information about a single step of the ma-

[3] We rejected the alternative of allowing an assertion to be accepted and executed if the asserter wins the challenge game, in order to prevent attacks where a malicious challenger deliberately loses the challenge game in order to get a false assertion accepted. The design we chose ensures that a challenger who deliberately loses will lose half their deposit to the miners (and the other half to the asserter with whom the challenger might be colluding), but a malicious challenger will not be able to force the acceptance of an invalid assertion.

chine's execution but the vast majority of the machine's state remains opaque to the Verifier. In section 4.4, we show that we can even eliminate this leak by doing the one step verification in a privacy-preserving manner.

Arbitrum's privacy is no coincidence, but rather a direct result of its model. Since the Arbitrum Verifier (e.g., the miners in a Nakamoto consensus model) do not run a VM's code, they do not need to see it. By contrast, in Ethereum, or any system that attempts to achieve "global correctness," all code and state has to be public so that anyone can verify it, and this model is fundamentally at odds with private execution.

Flexibility. Unanimous assertions provide a great deal of flexibility as managers can choose to reset a machine to any state that they wish and take any actions that they want (provided that the machine has the funds) – even if they are invalid by the machine's code. This requires unanimous agreement by the managers, so if any one manager is honest, this will only be done when the result is one that an honest manager would accept–such as winding down a VM that has gotten into a bad state due to a software bug.

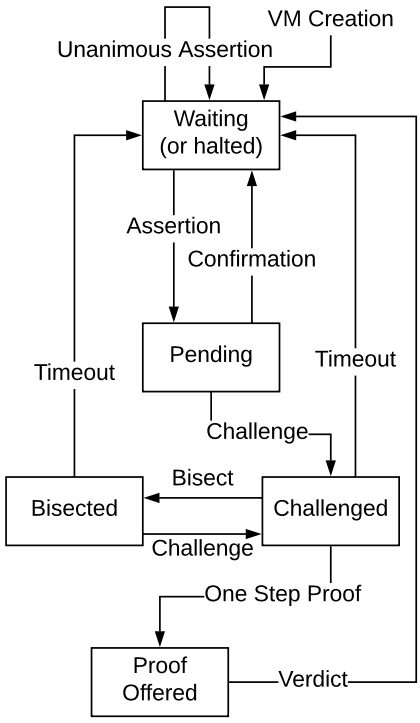

Figure 1: **Overview of the state machine that governs the status of each VM in the Arbitrum protocol.**

4 Arbitrum Design Details

This section describes the Arbitrum protocol and virtual machine design in more detail. The protocol governs the public process that manages and advances the public state of the overall system and each VM. The VM architecture governs the syntax and semantics of Arbitrum programs that run within a VM.

4.1 The Arbitrum Protocol

Arbitrum uses a simple cryptocurrency design, augmented with features to allow the creation and use of Virtual Machines (VMs), which can embody arbitrary functionality. VMs are programs running on the Arbitrum Virtual Machine Architecture, which is described below.

The Arbitrum protocol recognizes two kinds of actors: keys and VMs. A key is identified by (the cryptographic hash of) a public key, and the actor is deemed to have taken an action if that action is signed by the corresponding private key. The other kind of actor is a VM, which takes actions by executing code. Any actor can own currency. Arbitrum tracks how much currency is owned by each actor.

A VM is created using a special transaction type. The VM-creation transaction specifies a cryptographic hash of the initial state of the VM, along with some parameters of the VM, such as the length of the challenge period, the amounts of various payments and deposits that parties will make as the protocol executes further, as well as a list of the VM's managers.

For each VM, the Verifier tracks the hashed state of that VM, along with the amount of currency held by the VM, and a hash of its inbox. A VM's state can be changed via assertions about the VM's execution, which specify (1) the number of instructions executed by the VM, (2) the hash of the VM's state after the execution, and (3) any actions taken by the VM such as making payments. Further, the assertion states a set of preconditions that must be true before the assertion which specify (1) the hash of the VM's state before the execution, (2) an upper and lower bound on the time that the assertion is included in a block, (3) a lower bound on the balance held by the VM, and (4) a hash of the VM's inbox. The rules of Arbitrum dictate under which conditions an assertion is accepted. If an assertion is accepted, then the VM is deemed to have changed its state, and taken publicly visible actions, as specified by the assertion.

In the simplest case, an assertion is signed by all of the VM's managers. In this case, the assertion is accepted by the miners if the assertion is eligible, that is, if (1) the assertion's precondition matches the current state of the VM, (2) the VM is not in a halted state, and (3) the VM has enough funds to make any payments specified by the assertion. Unanimous assertions are relatively cheap

for verifiers to verify, requiring only checking eligibility and verifying the managers' signatures, so they require a small transaction fee.

In a more complicated case, an assertion is signed by just one of the managers–a "disputable assertion." Along with the assertion, the asserting manager must escrow a deposit. Such a disputable assertion is not accepted immediately, but rather, if it is eligible, it is published as pending, and other managers are given a pre-specified time interval in which they can challenge the assertion. (The number of steps allowed in a disputable assertion is limited to a maximum value that is set as a parameter when the VM is created, to ensure that other managers have enough time to emulate the declared number of steps of execution before the challenge interval expires.) If no challenge occurs during the interval, then the assertion is accepted, the VM is deemed to have made the asserted state change and taken the asserted actions, and the asserting manager gets its deposit back.

4.2 The Bisection Protocol

If a manager challenges an assertion, the challenger must escrow a deposit. Now the asserter and the challenger engage in a game, via a public protocol, to determine who is incorrect. The party who wins the game will recover its own deposit, and will take half of the losing party's deposit. The other half of the loser's deposit will go to the Verifier, as compensation for the work required to referee the game.

The game is played in alternating steps. After a challenge is lodged, the asserter is given a pre-specified time interval to bisect its previous assertion. If the previous assertion involved N steps of execution in the VM, then the two new assertions must involve $\lfloor N/2 \rfloor$ and $\lceil N/2 \rceil$ steps, respectively, and the two assertions must combine to be equivalent to the previous assertion. If no valid bisection is offered within the time limit, the challenger wins the game. After a bisection is offered, the challenger must challenge one of the two new assertions, within a pre-specified time interval.

The two players alternate moves. At each step, a player must move within a specified time interval, or lose the game. Each move requires the player making the move to make a small additional deposit, which is added to the stakes of the game.

After a logarithmic number of bisections, the challenger will challenge an assertion that covers a single step of execution. At this point the asserter must offer a one-step proof, which establishes that in the asserted initial state, and assuming the preconditions, executing a single instruction in the VM will reach the asserted final state and take the asserted publicly visible actions, if any. This one-step proof is verified by the Verifier. See Figure 1 for an overview of the state machine implementing this protocol.

4.3 The Arbitrum VM Architecture

The Arbitrum VM has been designed to make the Verifier's task of checking one-step proofs as fast and simple as possible. In particular, the VM design guarantees that the space to represent a one-step proof and the time to generate and verify such a proof are bounded by small constants, independent of the size and contents of the program's code and data.

As an example of an architectural choice to support constant-bounded proofs, the AVM does not offer a large, flat memory space. Providing an efficiently updatable hash of a large flat memory space would require the space to be hashed in Merkle Tree style, with a prover needing to provide Merkle proofs of memory state, which requires logarithmic proof space and logarithmic time to prove and verify. Instead, the Arbitrum VM provides a *tuple* data type that can store up to eight values, which can contain other tuples recursively. This allows the same type of tree representation to be built, but it is built and managed by Arbitrum code running in an application within the VM. With this design, reading or writing a memory location requires a logarithmic number of constant-time-provable Arbitrum instructions (instead of a single logarithmic-time provable instruction). The Arbitrum standard library provides a large flat memory abstraction for programmers' convenience.

We provide an overview of the VM architecture here. For a more detailed specification, see the extended version of this paper.

Types The Arbitrum VM's optimized operation is fundamentally dependent on its type system. In our prototype, types include: a special null value *None*, booleans, characters (i.e., UTF-8 code points), 64-bit signed integers, 64-bit IEEE floating point numbers, byte arrays of length up to 32, and tuples. A tuple is an array of up to 8 Arbitrum values. The slots of a tuple may hold any value, including other tuples, recursively, so that a single tuple might contain an arbitrarily complex tree data structure. All values are immutable, and the implementation computes the hash of each tuple when it is created, so that the hash of any value can be (re-)computed in constant time.[4]

VM State The state of a VM is organized hierarchically. This allows a hash of a VM's state to be computed

[4]Tuples, and by extension types, are a fundamental aspect of our VM design. Other non-crucial elements may change. For example, fewer types might be supported, such as only tuple and integer types.

in Merkle Tree fashion, and to be updated incrementally. The state hash can be updated efficiently as the machine's state changes, because the VM architecture ensures that instructions can only modify items near the root of the state tree and that each node of the state tree has a degree of no more than eight.

The state of a VM contains the following elements:

- an instruction stack, which encodes the current program counter and instructions (as described below);
- a data stack[5] of values;
- a call stack, used to store the return information for procedure calls;
- a static constant, which is immutable; and
- a single mutable register which holds one value.

When a VM is initialized, the instruction stack and static constant are initialized from the Arbitrum executable file; the data and call stacks are both empty; and the register is *None*. Note that because a single value can hold an arbitrary amount of data through recursive inclusion of tuples, the static constant can hold arbitrary amounts of constant data for use in a program, and the single register can be used to manage a mutable structure containing an arbitrary amount of data. Many programmers will choose to use a flat memory abstraction, built on top of such a mutable structure, such as the one provided in the Arbitrum standard library.

Instructions The VM uses a stack-based architecture. VM instructions exist to manipulate the top of the stack, push small integers onto the stack, perform arithmetic and logic operations at the top of the stack, convert between types, compute the hash of a value, compute a subsequence of a byte array, and concatenate byte arrays. Control flow instructions include conditional jump, procedure call, and return. Instructions to operate on tuples include an instruction to a create new tuple filled with *None*, to read a slot from a tuple, and to copy a tuple while modifying the value of one slot. Finally, there are instructions to interact with other parties, which are described below.

The Instruction Stack Rather than using a conventional program counter, Arbitrum maintains an "instruction stack" which holds the instructions in the remainder of the program. Rather than advancing the program counter through a list of instructions, the Arbitrum VM pops the instruction stack to get the next instruction to execute. (If the instruction stack is empty, the VM halts.) Jump and procedure call instructions change the instruction stack, with procedure call storing the old instruction stack (pushing a copy of the instruction stack onto the call stack) so that it can be restored on procedure return.

This approach allows a one-step proof to use constant space and allows verification of the current instruction and the next instruction stack value in constant time. [6]

Because a stack can be represented as a linked list, AVM implementations will likely follow our prototype implementation by arranging all of the instructions in a program into a single linked list and maintaining the instruction stack value as a pointer into that linked list.

The Assembler and Loader The Arbitrum assembler takes a program written in Arbitrum assembly language and translates it into an Arbitrum executable. The assembler provides various forms of syntactic sugar that make programming somewhat easier, including control structures such as if/else statements, while loops, and closures. The assembler also supports inclusion of library files, such as those in the standard library.

The Standard Library The standard library is a set of useful facilities written in Arbitrum assembly code. It contains about 3000 lines of Arbitrum assembly code, and supports useful data structures such as vectors of arbitrary size, key-value stores, an abstraction of a flat memory space on top of the register, and handling of time and incoming messages.

Interacting with other VMs or keys A VM interacts with other parties by sending and receiving messages. A message consists of a value, an amount of currency, and the identity of the sender and receiver. The `send` instruction takes values from the top of the stack and sends them as a message. If the message is not valid, for example because it tries to send more currency than the VM owns, the invalid message will be discarded rather than sent. A program uses the `inbox` instruction to copy the machine's message inbox to the stack. The standard library contains code to help manage incoming messages including tracking when new messages arrive and serving them one by one to the application.

The `balance` instruction allows a VM to determine how much currency it owns, and the `time` instruction al-

[5] A stack is represented as either *None*, representing an empty stack, or a 2-tuple *(top, rest)* where *top* is the value on top of the stack and *rest* is the rest of the stack, in the same format.

[6] A more conventional approach would keep an integer program counter, a linear array of instructions, and a pre-computed Merkle tree hash over the instruction array. Then a one-step proof would use a Merkle-tree proof to prove which instruction was under the current program counter. This would require logarithmic (in the number of instructions) space and logarithmic checking time for a one-step proof. By contrast our approach requires constant time and space.

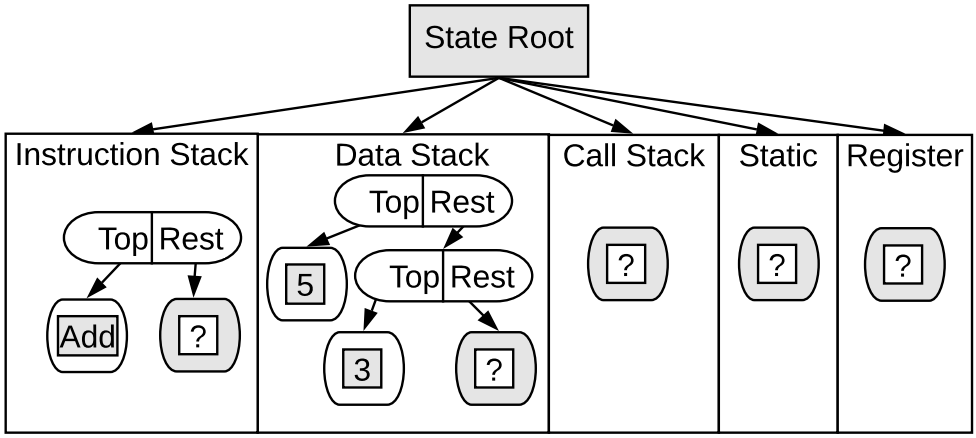

Figure 2: **Information revealed in a one step proof of an add instruction.** Outer boxes rounded represent value hashes and inner square boxes represent the values themselves. Gray boxes are values that are sent by the asserter to the verifier in the one-step proof.

lows a VM to get upper and lower bounds on the current time.

Preconditions, Assertions, and One-Step Proofs As described above, an assertion is a claim about an interval of a VM's execution. Each assertion is accompanied by a set of *preconditions* consisting of: a hash of the VM's state before the asserted execution, a hash of the VM's inbox contents, an optional lower bound on the VM's currency balance, and optional lower and upper bounds on the time (measured in block height). An assertion will be ignored as ineligible unless all of it preconditions hold. (Parties may choose to store an ineligible assertion in the hope that it becomes eligible later.)

In addition to preconditions, an assertion contains the following components: the hash of the machine state after the execution, the number of instructions executed, and the sequence of messages emitted by the VM.

The Arbitrum protocol may require a party to provide a one-step proof, which is a proof of correctness, assuming a set of preconditions, for an assertion covering the execution of a single instruction. A one-step proof must provide enough information, beyond the preconditions, to enable the Verifier to emulate the single instruction that will be executed. Because the state of the VM is organized as a Merkle Tree, and the starting state hash of the VM, which is just the root hash of that Merkle Tree, is given as a precondition, the proof need only expand out enough of the initial state Merkle tree to enable the Verifier to emulate execution of the single instruction, compute the unique assertion that results from executing that one instruction given the preconditions, and verify that it matches the claimed assertion.

A one-step proof expands out any parts of the state tree that are needed by the Verifier. For example, suppose that the instruction to be executed pops an item off the stack. Recall that the stack is represented as *None* for the empty stack, and otherwise as a 2-tuple *(top, rest)* where *top* is the top item on the stack and *rest* is the rest of the stack. In this example, if the stack hash is equal to the hash of *None*, then the Verifier will know that the stack is empty. Otherwise the prover will need to provide the hashes of *top* and *rest*, allowing the Verifier to check that those two hashes combine to yield the expected stack hash. Similarly, if the instruction is supposed to add two values, and the Verifier only has the hashes of the values, the proof must include the two values. In all cases the prover provides values that the Verifier will need to emulate the specified instruction, and the Verifier checks that the provided values are consistent with the hashes that the Verifier has already received. The Arbitrum VM emulator used by the prover automatically determines which elements must be provided in the proof. See Figure 2 for an illustration of the information revealed to a Verifier during a one step proof of an add instruction.

Messages and the Inbox Messages can be sent to a VM in two ways: a key can send a message by putting a special message delivery transaction on the blockchain; and another VM can send a message by using the send instruction. A message logically has four fields: data, which is an AVM value (marshaled into a byte array on the blockchain); a non-negative amount of currency, which is to be transferred from the sender to the receiver; and the identities of the sender and receiver of the message.

Every VM has an inbox whose hash is tracked by the Verifier. An empty inbox is represented as the AVM value *None*. A new message M can be appended to a VM's inbox by setting the inbox to a 2-tuple *(prev, M)*,

where *prev* is the previous state of the inbox. A VM can execute the `inbox` instruction which pushes the current value of the VM's inbox onto the VM's stack.

A VM's managers track the state of its inbox, but the Verifier only needs to track the hash of the inbox, because that is all that will be needed to verify a one-step proof of the VM receiving the inbox contents. If the VM later processes the inbox contents, and a one-step proof of some step of that processing is needed, the managers will be able to provide any values needed.

Because the `inbox` instruction gives the VM an inbox state that may be a linked list of multiple messages, programmers may wish to buffer those messages inside the VM to provide an abstraction of receiving one message at a time. The Arbitrum standard library provides code to do this as well as track when new messages have arrived in the inbox.

4.4 Extensions

In this section, we describe extensions to Arbitrum's design that may prove useful, particularly when the Arbitrum Verifier is implemented as a public blockchain.

Off-chain progress Arbitrum allows VMs to perform orders of magnitude more computation than existing systems at the same on-chain cost. However, usage of VMs frequently depends on communication between a VM's managers and the VM itself. In our prior description of Arbitrum's protocol, this communication had to be on-chain and thus was limited by the speed of the consensus mechanism. Arbitrum is compatible with state-channel and sidechain techniques, and there are several constructions that allow managers to communicate with a VM and unanimously advance a VM's state off-chain. We present details of one such construction in the extended version of this paper.

Zero Knowledge one step proofs While Arbitrum has good privacy properties, there is one scenario in which a small privacy leak is possible. A manager submitting a one step proof will be forced to reveal some of the state as part of the proof. While only a small portion of the state will be revealed for each challenge, and only if the managers fail to agree on a unanimous assertion, this can potentially be sensitive data.

We can instead implement the one step proof as a zero-knowledge protocol using Bulletproofs [7]. To do so will require encoding a one step VM transition as an arithmetic circuit and proving that the transition is valid. While we could use SNARKs [4, 16, 27], Bulletproofs have the benefit that they do not require a trusted setup. Although verification time for Bulletproofs is linear in the circuit, considering that a one-step transition circuit will be small, and that one-step proofs will be infrequent events, this should not be a problem in practice.

While zero-knowledge proofs can in theory be used to prove the correctness of the entire state transition (and not just a single step), doing this for complex computations is not feasible with current tools. Combining the challenge and bisection protocol with a zero-knowledge proof only at the last step allows us to simultaneously achieve scalability and full privacy. This takes advantage of the fact that the Arbitrum VM is designed to simplify one-step proofs.

Reading the Blockchain In our current design, Arbitrum VMs do not have the ability to directly read the blockchain.

If launched as a public blockchain, we could easily extend the VM instruction set to allow a VM to read the blockchain directly. To do so, we would create a canonical encoding of a block as an Arbitrum tuple, with one field of that tuple containing the tuple representing the previous block in the blockchain. This would allow a VM that had the tuple for the current block to read earlier blocks. The precondition of an assertion would specify a recent block height, and the VM would have a special instruction that pushes the associated block tuple to the stack. In order to be able to verify a one-step proof of this instruction, the Verifier just needs to keep track of the Arbitrum tuple hash of each block (just a single hash per block).

We stress that reading the blockchain does not require putting lots of data on a VM's data stack. A blockchain read consists of putting just the top-level tuple of the specified block on the stack. To read deeper into the blockchain, this tuple can be lazily expanded, providing the VM with just the data that it needs to read the desired location.[7]

[7] Note that reading the blockchain in this manner supports *oblivious reads* compatible with zero-knowledge proofs, as the Verifier does not need to know what position (if any) in the blockchain is being read. The Verifier need only verify the top-level tuple hash, which is the hash of a recent block. If the tuple was expanded to read deeper into the blockchain, this all happens inside Arbitrum application code and the location of the read will not be published on-chain. In this manner, blockchain reads are fully compatible with zero-knowledge one-step proofs. In particular, the Verifier would always provide the specified block tuple hash as an input to the zero-knowledge proof. If indeed the one-step proof is on a read-blockchain instruction, the proof would verify that the correct hash was put on the stack. The zero knowledge proof would not leak information as to whether the blockchain was actually read (as the block hash is always an input to the proof even if no read occurred) or where on the blockchain a read occurred (since the current block tuple could have been expanded inside Arbitrum application code to read anywhere in the blockchain).

5 Implementation and Benchmarks

In order to refine and evaluate Arbitrum, we produced a full implementation of the Arbitrum system. This includes code to represent all parties involved: a centralized Verifier, a VM, an honest manager, and a key-based actor. These parties are fully capable of performing all parts of the Arbitrum protocol. Our implementation comprises about 6800 lines of Go code, including about 3400 lines for the VM emulator, 1350 lines for the assembler and loader, 650 lines for the honest manager, 550 lines for the Verifier, and the remainder for various shared code.

In order to ease the coding of more powerful smart contract VMs, we implemented the Arbitrum standard library which contains about 3000 lines of Arbitrum assembly code, supporting useful data structures such as large tuples, key-value stores, queues, and character strings; and utilities for handling messages, currency, and time.

We demonstrate the power and versatility of this implementation by implementing two smart contracts.

5.1 Escrow Contract

We first discuss a simple escrow contract. The escrow code first waits for a message containing the identities of three parties (Alice, Bob, and Trent) and an integer deadline, along with some amount of currency that the VM will hold. The VM then waits for a message from Trent, ignoring messages that arrive from anybody else. If the message from Trent contains an even integer, the VM sends the currency to Alice and halts. If the message from Trent contains something else, the VM sends the currency to Bob and halts. If the current time exceeds the deadline, the VM sends half of the currency to Alice, the remaining currency to Bob, and then halts. This requires 59 lines of Arbitrum assembly code, which makes significant use of the standard library. The executable file produced by the assembler contains 4016 instructions.

Executing the contract requires 5 total transactions to be added to the blockchain. The initial create VM transaction is 309 bytes. After that a 310 byte message is sent to the VM communicating the identities of the parties involved and the deadline, and giving currency to the VM. Next, Trent indicates his verdict by sending a 178 byte message to the VM.

Next, the VM must be executed to actually cause the payouts. First a 350 byte assertion is broadcast, asserting the execution of 2897 AVM instructions, leaving the VM in the halted state. Next after the challenge window has passed, a confirmation transaction of 113 bytes is broadcast confirming and accepting the asserted execution. The entire process requires a total of 1,260 bytes to be written to the blockchain.

5.2 Iterated Hashing

One area where Arbitrum shines is the efficiency with which it can carry out VM computation. To demonstrate this, we measured the throughput of an Arbitrum VM which performs iterative SHA-256 hashing. The code for this VM is an infinite loop where the VM hashes 1000 times and then jumps back to the beginning. The VM code makes use of the AVM's hash instruction, which is implemented in native code.

We evaluated operating performance of this VM on an early 2013 Apple MacBook Pro, 2.7GHz Intel Core i7. As a baseline, using native code on the same machine, we were able to perform 1,700,000 hashes per second. Running the VM continuously we were able to advance the VM by 970,000 hashes per second. Our implementation was able to achieve over half of the raw performance of native code. This stands in comparison to Ethereum, which is capable of processing a total of approximately 1600 hashes per second (limited by Ethereum's global gas limit, which is required due to the Verifier's Dilemma).

Arbitrum's performance advantage extends further. While we demonstrated the current limit on execution inside a single VM, the Verifier is capable of handling large numbers of VMs simultaneously. Instantiating many copies of the Iterated Hashing VM, we measured that the Verifier node running on our machine was capable of processing over 5000 disputable assertions per second. This brings the total possible network throughput up to over 4 billion hashes per second, compared to 1600 for Ethereum.

6 Background and related work

6.1 Refereed Delegation

The problem of delegating computation involves a resource-bounded client outsourcing computation to a more powerful server. The server should provide a proof that it correctly carried out the computation, and checking the proof should be far more efficient for the verifier than performing the computation itself [17].

Refereed-delegation (RDoC) is a two-server protocol for the problem of delegating computation [10, 11]. The computation is delegated to multiple servers that independently report the result to the client. If they agree, the client accepts the result. If the servers disagree, however, they undergo a bisection protocol to identify a one-step disagreement. The client can then efficiently evaluate the single step to determine which server was lying. Aspects of Arbitrum's bisection protocol are very

similar to RDoC. In Arbitrum, it is as if the Verifier is outsourcing a VM's computation back to the VM's managers, who in many cases are the parties interested in the VM's computation. Arbitrum's VM architecture makes dispute resolution very efficient.

6.2 Bitcoin

Bitcoin is a decentralized digital currency [26].

Bitcoin natively supports only a simple scripting language that is not Turing Complete and is mainly used for signature validation. Many techniques have been developed to allow more complex scripting on top of Bitcoin's scripting language. These generally fall into two categories: (1) protocols that use cryptographic tools to enable more complex functionality while restricting themselves to Bitcoin's scripting language, and (2) protocols that use Bitcoin as a consensus layer, including raw data on the blockchain with additional validation rules known by nodes running the protocol, but not validated by the Bitcoin miners.

The first variety of scripting enhancements include zero-knowledge contingent payments [3, 9, 23] that are able to realize a fair exchange of digital goods. While powerful and efficient, zero-knowledge contingent payments are limited and unable to realize general smart contracts. The latter variety, which includes Counterparty [1] and Open Assets [12], pushes the entire effort of validation onto every wallet. In these overlay protocols, every node must validate every transaction (even those that they are not a part of) in order to have confidence in correctness. Contrast this to Arbitrum in which miners guarantee the correctness of all monetary transactions, and nodes must only monitor the internal state of the VMs they care about.

6.3 Ethereum

Ethereum [31] is a digital currency that supports stateful, Turing-complete smart contracts. Miners emulate a contract's code and update the state accordingly. In order for an Ethereum block to be valid, miners must correctly emulate all of the contract computations that they include in their block and correctly update the state (including monetary balances) to reflect those changes. If a miner does not update the state correctly, other miners will reject that block.

Ethereum aims for "global correctness," or the ability of every participant in the system to trust that every contract has been correctly executed contingent only on the mining consensus process working as intended. In contrast, Arbitrum does not try to provide correctness guarantees for a VM to parties who are not interested in that VM, and this enables Arbitrum to reap large advantages in scalability and privacy. In Arbitrum, parties can safely ignore VMs that they are not interested in.

Limitations of Ethereum style smart contracts

Ethereum's approach to smart contracts has several drawbacks.

Scalability. It has long been known that Ethereum's model cannot scale. Requiring miners to emulate every smart contract is expensive, and this work must be duplicated by every miner. While Ethereum does require the parties who are interested in a computation to compensate miners (with "gas") for the cost of executing, this does not lower the cost – it only shifts it.

Ethereum copes with the Verifier's Dilemma by having a "global gas limit" that severely limits the amount of computation that can be included in each block.[8] Ethereum's global gas limit is a significant limitation that makes many computations – that would take just seconds to execute on a modern CPU – unachievable [8, 24]. Even for computations which are below the gas limit, Ethereum's pay-per-instruction model can become prohibitively expensive.

Privacy. All Ethereum contract code is public, and this is a necessity of the model as every miner needs to be able to emulate all of the code. Any privacy in Ethereum must come as an overlay. There has been progress toward using zkSNARKs [4, 16, 27] in Ethereum so that miners can verify proofs while inputs to the contract call remain hidden. However, the ability to do this this is severely limited in practice as the cost to verify a SNARK is high,[9] so the throughput would be severely limited to just a few such transactions per block. Moreover, SNARKs impose a heavy computational cost on the prover.

Inflexibility. In legal contracts, the parties to a contract can modify or cancel the contract by mutual agreement. This is considered an important feature of legal contracts, because it prevents the parties from being trapped by an erroneous contract or unforeseen circumstances. For Ethereum-style smart contracts, deviation from the code

[8] While Arbitrum does limit the number of steps of computation in an assertion in some cases, Arbitrum's limit is much less constraining. The Arbitrum limit applies only to disputable assertions, not to unanimous assertions which can include an unlimited number steps. Also, Arbitrum's limit, when it applies, is per VM and assumes many VMs can be managed in parallel, whereas Ethereum's is a global limit on the total computation over all VMs.

[9] A transaction on the Ethereum testnet (0x15e7f5ad316807ba16fe669a07137a5148973235738ac424d5b70fk89ae7625e3) validated a SNARK using 1,933,895 gas. At the current mainnet gas limit of 7,976,645, this would only allow 4 transactions per block.

is not possible. In Arbitrum, a modification to a contract VM is possible, as long all of the VM's honest managers will agree to it.

6.4 Other proposed solutions

We now discuss other proposed solutions for smart contract scalability and/or privacy and compare them with Arbitrum.

Zero-knowledge proofs. Hawk [18] is a proposed system for private smart contracts using zkSNARKs [16, 27]. Hawk has strong privacy goals that include hiding the amounts and transacting parties of monetary transfers, hiding contract state from non-participants, and supporting private inputs that are hidden even from other participants in the contract. However, Hawk suffers several drawbacks that make it infeasible in practice. Firstly, SNARKs require a per-circuit trusted setup, which means that for every distinct program that a contract implements, a new trusted setup is required. While multi-party computation can be used to reduce trust in the setup, this is infeasible to perform on a per-circuit basis as is required by Hawk. Secondly, Hawk does not improve scalability as each contract requires kilobytes of data to be put on-chain. Finally, privacy in Hawk relies on trusting a third-party manager who gets to see all the private data.

Trusted Execution environments (TEEs). Several proposals [6, 13, 20, 33] would combine blockchains with trusted execution environments such as Intel SGX. Ekiden [13] uses a TEE to achieve scalable and private smart contracts. Whereas Arbitrum hides the code and state of a smart contract from external parties, Ekiden hides the state from external parties and also allows parties of a contract to hide private inputs from one another.

The drawback of Ekiden and systems that rely on TEEs more generally is the additional trust required for both privacy as well as the correctness of contract execution. This includes both trusting that the hardware is executing correctly and privately as well as trusting the issuer of the attestation keys (*e.g.*, Intel).

Secure Multiparty Computation. Secure multiparty computation is a cryptographic technique that allows parties to compute functions on private inputs without learning anything but their output [21]. Several works have proposed to incorporate secure multiparty computation onto blockchains [2, 19, 34]. This enables attaching monetary conditions to the outcome of computations and incentivizing fairness (by penalizing aborting parties).

Unlike Arbitrum which can make progress even when nodes go offline, MPC based systems require the active (and interactive) participation of all computing nodes. Even with recent advances in the performance of secure-multiparty computation, the cryptographic tools impose a significant efficiency burden.

Scalability via incentivized verifiers. Several proposals (*e.g.*, [30, 32]) have separate parties (other than the miners) perform verification of computation, but depending on how verifiers are rewarded, these results may fall victim to the Participation Dilemma.

The most popular of these systems is TrueBit [30]. Unlike Arbitrum, TrueBit is stateless and not a standalone system. TrueBit provides a mechanism for an Ethereum contract to outsource computation and receive the result at a cost to the contract that is lower than Ethereum's gas price. In TrueBit, third-party Solvers perform computational tasks and their work is checked by third-party Verifiers (which play a different role than Arbitrum verifiers). TrueBit Verifiers can dispute the results given by the Solver, and disputes are settled via a challenge-response protocol similar to the one used in Arbitrum.

TrueBit attempts to achieve global correctness by incentivizing TrueBit Verifiers to check computation and challenge incorrect assertions. To participate, TrueBit Verifiers must put down a deposit, which they will lose if they falsely report an error. In order to incentivize verifiers to participate, the TrueBit protocol occasionally introduces deliberate errors and TrueBit Verifiers collect rewards for finding them.

If m TrueBit Verifiers find the same error, they split the reward using a function of the form $f_c(m) = c \cdot 2^{-m}$. As shown in Section 2.3, this is One-Shot Sybil-Proof. However, since it is a participation game, they are susceptible to the Participation Dilemma, and by Theorem 1, TrueBit admits an equilibrium in which there is only a single TrueBit Verifier (using multiple Sybils), and if this occurs, this verifier can cheat at will.

Although they don't formally analyze it, TrueBit acknowledges this type of attack and proposes some ad-hoc defenses. First, they assume that a single verifier will not have enough money to make the deposits needed to successfully bully out all other verifiers. While this assumption may be helpful, it is not clear that it holds, and in particular multiple adversaries could pool their funds to launch this attack. (Note that an attacker would not forfeit these funds in order to execute this attack, but would just need to have them on hand.)

Even if the assumption does hold, it is still possible for an adversary to bully out all other verifiers from a particular contract by verifying the contract with multiple Sybils. To defend against this, TrueBit proposes a "default strategy" in which verifiers choose at random which task to verify, and do not take into account the

number of verifiers to previously verify a contract. This proposal is problematic, however, as the default strategy is dominated: instead of choosing where to verify randomly, a verifier is better off if it chooses the tasks with fewer additional verifiers. Not only is following the "default strategy" not an equilibrium, but is dominated by a better strategy, no matter what the others do.

TrueBit also does not provide privacy as it allows anybody to join the system as a verifier, and thus anybody must be able to learn the full state of any VM.

Another key difference between TrueBit and Arbitrum is that in TrueBit, the cost for computation is linear in the number of steps executed. For every computational task performed in TrueBit, the party must pay a tax to fund the solving and verification of that task. The TrueBit paper estimates that this tax is between 500%-5000% of the actual cost of the computation. Although the cost of computation in TrueBit is lower than the cost in Ethereum, it still suffers from a linear cost.

TrueBit proposes to use Web Assembly for the VM architecture. However, unlike the Arbitrum Virtual Machine which ensures that one-step proofs will be of small constant size, Web Assembly has no such guarantee.

Plasma. Plasma [28] attempts to achieve scaling on top of Ethereum by introducing the concept of child-chains. Child-chains use their own consensus mechanism to choose which transactions to publish. This consensus mechanism enforces rules which are encoded in a smart contract placed in Ethereum. If a user on the child-chain believes that the child-chain has behaved incorrectly or maliciously, they can submit a fraud proof to the contract on the main chain in order to exit the child-chain with their funds.

This approach suffers from a number of problems. Firstly, similarly to sharding, Plasma child-chains each exist in their own isolated world, so interaction between people on different child-chains is cumbersome. Secondly, the details of how complex fraud proofs could actually be constructed inside a Plasma contract are lacking. Plasma contracts need to somehow specify all of the consensus rules and ways to prove fraud on a newly defined blockchain which is a complex and currently unsolved problem inside an Ethereum contract. Finally, moving data out of the main blockchain creates data availability challenges since in order to generate a fraud proof you must have access to the data in a Plasma block and there is no guaranteed mechanism for accessing this data. Because of this issue, Plasma includes many mitigations which involve users exiting a Plasma blockchain if anything goes wrong.

Due to the complexities of implementing Plasma child-chains with smart contract capabilities like Ethereum, all current efforts to implement Plasma use simple UTxO based systems without scripting in order allow simple proofs. Plasma proposes using TrueBit as a sub-component for efficient fraud proofs in child chains with smart contracts, but as mentioned TrueBit uses an off-the-shelf VM which does not give guarantees on proof size or efficiency. Indeed, Plasma may benefit from using the Arbitrum Virtual Machine.

State Channels. State channels are a general class of techniques which improve the scalability of smart contracts between a small fixed set of participants. Previous state channel research [5, 14, 15, 25] has mainly focused on a different type of scaling than Arbitrum has achieved. Arbitrum allows on-chain transactions with a very large amount of computation and state, with low cost. State channels allow a set of parties to mutually agree to a sequence of messages off-chain and only post a single aggregate transaction after processing them all.

State channel constructions focus on the optimistic case where all parties are honest and available, but fail to work smoothly and efficiently in other situations. Specifically, state channels must be prepared to resolve on-chain if any member of the channel refuses or is unable to continue participating. This on-chain resolution mechanism requires the execution of an entire state transition on-chain. Thus, state channels are limited to only doing computation that the parties could afford to do on-chain, since otherwise dispute resolution will be infeasible. Arbitrum is still efficient even if managers are not all active at all times, or if there are disputes.

7 Conclusion

We have presented Arbitrum, a new platform for smart contracts with significantly better scalability and privacy than previous solutions. Our solution is consensus agnostic and is pluggable with any existing mechanism for achieving consensus over a blockchain. Arbitrum is elegant in its simplicity, and its straightforward and intuitive incentive structure avoids many pitfalls that affect other proposed systems.

Arbitrum creates incentives for parties to agree off-chain on what smart contract VMs will do, and even if parties act contrary to incentives the cost to miners or other verifiers is low. Arbitrum additionally uses a virtual machine architecture that is custom-designed to reduce the cost of on-chain dispute resolution. Moving the enforcement of VM behavior mostly off-chain, and reducing the cost of on-chain resolution, leads to Arbitrum's advantages in scalability and privacy.

8 Acknowledgements

Steven Goldfeder is supported by an NSF Graduate Research Fellowship under grant DGE 1148900. S. Matthew Weinberg is supported by NSF grant CCF-1717899.

References

[1] Counterparty protocol specification. `https://counterparty.io/docs/protocol_specification/`, accessed: 2018-01-01

[2] Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on bitcoin. In: Security and Privacy (SP), 2014 IEEE Symposium on

[3] Banasik, W., Dziembowski, S., Malinowski, D.: Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In: European Symposium on Research in Computer Security. pp. 261–280. Springer (2016)

[4] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: Verifying program executions succinctly and in zero knowledge. In: Advances in Cryptology–CRYPTO 2013, pp. 90–108. Springer (2013)

[5] Bentov, I., Kumaresan, R., Miller, A.: Instantaneous decentralized poker. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 410–440. Springer (2017)

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

[7] Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Efficient range proofs for confidential transactions. Tech. rep.

[8] Bunz, B., Goldfeder, S., Bonneau, J.: Proofs-of-delay and randomness beacons in Ethereum. In: Proceedings of the 1st IEEE Security & Privacy on the Blockchain Workshop (April 2017)

[9] Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: Attacks and payments for services. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 229–243. ACM (2017)

[10] Canetti, R., Riva, B., Rothblum, G.N.: Practical delegation of computation using multiple servers. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 445–454. ACM (2011)

[11] Canetti, R., Riva, B., Rothblum, G.N.: Refereed delegation of computation. Information and Computation 226, 16–36 (2013)

[12] Charlon, F.: Open assets protocol (oap/1.0). Online, https://github.com/OpenAssets/open-assets-protocol/blob/master/specification.mediawiki (2013)

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

[14] Coleman, J.: State channels (2015)

[15] Dziembowski, S., Eckey, L., Faust, S., Malinowski, D.: Perun: Virtual payment channels over cryptographic currencies. Tech. rep., IACR Cryptology ePrint Archive, 2017: 635 (2017)

[16] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct nizks without pcps. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer (2013)

[17] Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. In: Proceedings of the fortieth annual ACM symposium on Theory of computing. pp. 113–122. ACM (2008)

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

[19] Kumaresan, R., Moran, T., Bentov, I.: How to use bitcoin to play decentralized poker. In: CCS

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

[21] Lindell, Y., Pinkas, B.: Privacy preserving data mining. In: Annual International Cryptology Conference. pp. 36–54. Springer (2000)

[22] Luu, L., Teutsch, J., Kulkarni, R., Saxena, P.: Demystifying incentives in the consensus computer. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 706–719. ACM (2015)

[23] Maxwell, G.: Zero knowledge contingent payments. URl: `https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment` (2011)

[24] McCorry, P., Shahandashti, S.F., Hao, F.: A smart contract for boardroom voting with maximum voter privacy. IACR Cryptology ePrint Archive 2017, 110 (2017)

[25] Miller, A., Bentov, I., Kumaresan, R., Cordi, C., McCorry, P.: Sprites and state channels: Payment networks that go faster than lightning

[26] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)

[27] Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: IEEE Symposium on Security and Privacy, 2013

[28] Poon, J., Buterin, V.: Plasma: Scalable autonomous smart contracts. White paper (2017)

[29] Roughgarden, T.: Lecture #5: Incentives in peer-to-peer networks. `http://theory.stanford.edu/~tim/f16/l/l5.pdf` (October 2016)

[30] Teutsch, J., Reitwiener, C.: A scalable verification solution for blockchains (2017)

[31] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper 151, 1–32 (2014)

[32] Wood, G.: Polkadot: Vision for a heterogeneous multi-chain framework (2017)

[33] Zhang, F., Daian, P., Kaptchuk, G., Bentov, I., Miers, I., Juels, A.: Paralysis proofs: Secure dynamic access structures for cryptocurrencies and more

[34] Zyskind, G., Nathan, O., Pentland, A.: Enigma: Decentralized computation platform with guaranteed privacy. arXiv preprint arXiv:1506.03471

A Participation Games: Full proof and discussion

First, we provide a proof of Theorem 1. To do this, we require a more formal setup than provided in Section 2.3. Every round, a participation game is played. Players have *time-discounted utilities* for some discounting parameter $\gamma < 1$. That is, the utility of round r is discounted at a rate of γ^r times the payoffs in the first round. Note that this is necessary in order for payoffs to be finite and the notion of best-responding to make sense. We will take $\gamma \to 1$. That is, the game is played for a fixed $\gamma < 1$, but we will consider the case where γ is very close to 1.

Definition 1 (One-Shot Sybil-Proof)**.** *We say that a participation game $f(\cdot)$ is **One-Shot Sybil-Proof** if for all k, $\ell \cdot f(k+\ell) \leq f(k+1)$. Note that this is equivalent to saying the strategy $s_i = 1$ is always a best response.*

Observation 1. *Every One-Shot Sybil-Proof participation game has $f(n+1) \leq f(n)/2$.*

Proof. Consider $\ell = 2$ in the definition of One-Shot Sybil-Proof. The claim immediately follows. □

Definition 2 (Participation Parameter)**.** *Define the **participation parameter** of a Sybil-proof participation game to be the maximum k such that $f(k) > 1$.*

Proof of Theorem 1. Let k be the participation parameter of the participation game. If $k = 1$, then it is trivially an equilibrium for player one to participate with $s_1 = 1$ every round, and all other players to not participate, and the theorem is proved.

If $k > 1$, we will consider any $1 > \gamma \geq 1 - \frac{1}{3kf(1)}$. Consider the following equilibrium:

- Player one participates and sets $s_1 = k$ in every round.
- Player $i \in [2,k]$ uses the following strategy: if during any of the previous $R = 12kf(1)^2$ rounds, $\sum_{j \neq i} s_j < k - i - 1$, set $s_i = 1$. Otherwise, set $s_i = 0$.
- Players $i > k$ set $s_i = 0$.

First, observe that all players $i > 1$ are best-responding, by definition of the participation parameter. Player one will set $s_1 = k$ every round no matter what, so all other players will set $s_j = 0$. Therefore, in any round the decisions faced by player i is simply whether to set $s_j = \ell$ and get reward $\ell \cdot f(k+\ell)$, without affecting anyone's strategies in any future rounds. By the fact that $f(\cdot)$ is One-Shot Sybil-Proof, we have that $\ell \cdot f(k+\ell) \leq f(k+1)$. By definition of the participation parameter, $f(k+1) \leq 1$. So player i would get reward at most 1 by participating, and have to pay cost 1, giving them non-positive utility by participating. Therefore, all players $i > 1$ are best responding (getting zero utility, but with no options that give higher utility).

Now, we wish to prove that player 1 is also best responding. Note that it is certainly possible for player 1

to improve their payoff in one round: they can achieve $\ell \cdot f(\ell)$ for any ℓ immediately after a round where they set $s_i = k$. Immediately from the definition of One-Shot Sybil-Proof, we see that player 1 would make more profit in this round by setting $s_i = 1$. However, this would cost them in future rounds, and it causes other players to participate.

Specifically, observe first that player 1 is strictly better off setting $s_1 = k$ in any round than $s_1 > k$. This is because all other players behave the same in every future round regardless of whether $s_1 = k$ or $s_1 > k$, and $s_1 = k$ yields strictly higher reward in the present round. So we need only consider deviations where $s_1 < k$.

Now consider the payoff of player 1 if they set $s_1 = k$ in every round. Each round they will get exactly $k \cdot f(k) - 1 := A$. So player 1 gets reward $\geq A/(1-\gamma)$.

Consider instead the maximum payoff if player 1 if they set $s_1 = \ell < k$ in some round. In this round, player 1 will get payoff $\ell \cdot f(\ell) - 1 > \varepsilon$. But now consider the subsequent R rounds, and call this set of rounds $\mathscr{R}$. In at most k of these rounds is it possible that $\sum_j s_j < k$. This is because $\sum_{j \neq 1} s_j \geq k - X$, where X is the minimum s_1 played over the previous rounds of $\mathscr{R}$. This is because if in *any* prior round in $\mathscr{R}$ we had $s_1 = X$, then players $2, \ldots, k - X + 1$ will all participate for the remaining rounds in $\mathscr{R}$. So the only way we can possibly have $\sum_j s_j < k$ is if $s_i < X$. As there are only k possible values to report, X can only decrease up to k times, meaning that there are at most k rounds where $\sum_j s_j < k$. Intuitively, what's going on is that every time player 1 lowers their Sybil count from the previous minimum, they get one awesome round where the total number of participants is $< k$. But all future rounds in $\mathscr{R}$ have increased participation from others, so the total participation will be at least k until player 1 further lowers their on Sybil count.

In each of these k rounds, player 1 might get a payoff of up to $f(1) - 1 = C$ (this is a very loose upper bound). However, in each of the other rounds, player 1 gets a payoff of at most $(k-1)f(k) - 1 \leq A - 1$. This is because there are at least k total participants in all other rounds, at least one of which is not player 1. So if player 1 is participating, the best case for them is that they are $k-1$ of the participants with only one other participant. So player 1's total payoff during these R rounds is upper bounded by:

$$\sum_{r=0}^{R-1} (A-1)\gamma^r + kf(1) = (A-1)(1-\gamma^R)/(1-\gamma) + kf(1)$$
$$= A(1-\gamma^R)/(1-\gamma) + kf(1) - (1-\gamma^R)/(1-\gamma).$$

Finally, observe that the total payoff for the entire remainder of the game from $R+1$ until it terminates is at most $\gamma^R \cdot f(1)/(1-\gamma)$. This is because the most value that can possibly be earned in round r is $\gamma^r f(1)$, so summing from $r = R$ to ∞ yields the above. This means that if the player deviates from $s_1 = k$ in round one, their total payoff is at most:

$$A/(1-\gamma) + kf(1) - (1-\gamma^R)/(1-\gamma) + \gamma^R f(1)/(1-\gamma).$$

Observe that the first term is exactly the reward achieved by setting $s_1 = k$ in every round. The added term can be made arbitrarily negative by setting γ, R appropriately. In particular, setting $\gamma = 1 - \frac{1}{3kf(1)}$, $R = 12kf(1)^2$ yields:

$$kf(1) - (1-\gamma^R)/(1-\gamma) + \gamma^R f(1)/(1-\gamma)$$
$$= kf(1) - 3kf(1) \cdot (1-\gamma^R) + \gamma^R \cdot 3kf(1)^2$$
$$= kf(1) \cdot \left(-2 + 3(f(1)+1)\gamma^R\right) < 0.$$

The final inequality follows because R is sufficiently large.

□

A quick comment on Theorem 1 is warranted. First, observe that our constants γ and R are really wasteful in order to keep the proof as simple as possible. Certainly we could optimize the constants, but this is not the point of the theorem. In addition, we of course are not claiming to predict that this is how players will behave in a participation game. There are numerous equilibria. The point we are making is that there are *provably bad equilibria* in the repeated game, despite the sound logic for one-shot reasoning, and these equilibria are quite (qualitatively) natural: most players react to the market, and one player cleverly stays one step ahead. Given this, and the very plausible existence of other undesirable equilibria, we would not predict that the one-shot sybil-proof equilibrium arises in the repeated game.

A.1 Discussion of possible defenses

In this section, we overview some "outside-the-box" defenses against the participator's dilemma. These defenses seem a) technically challenging (perhaps impossible) and b) costly - scaling linearly with the computational power of a possible adversary. The main idea is that our analysis of participation games considered one task in isolation where it was feasible for every player to participate in every round.

Consider instead a set of T participation games played in parallel, with the constraint that any player can simultaneously enter at most A of them. The bound A may come from limits on computational power, or required

monetary deposits. The "natural" state of affairs, however, would have $A > T$, reducing us back to the original participation game. That is, one should expect a single verifier (or conglomerate of verifiers) to have the computational power to process all contracts. Similarly, assuming that any ordinary participant can amass the funds for a deposit, a single wealthy verifier (or conglomerate) should certainly be able to amass the funds to deposit everywhere. So this approach initially doesn't seem to buy anything.

One potential avenue for defense is to introduce *dummy contracts* that are indistinguishable from the rest, to artificially inflate $T > A$. The downside to this is that if dummy contracts are to be indistinguishable from the rest, they must also reward verifiers, and therefore the cost of the system will blow up. Even if one is willing to pay the cost, this solution has some pitfalls:

- It's unclear how to design dummy transactions that are truly indistinguishable from the rest.
- Even if dummy transactions are indistinguishable from the rest, an adversary could still try to flood verification of a specific contract they're invested in, encouraging others to spend their limited deposits/computational power verifying elsewhere.

If somehow one is able to bypass the above problems, the cost of implementing dummy contracts grows linearly with the ratio A/T (where T is the natural desired throughput). We include the results of some simulations confirming this below.

With enough dummy transactions, the game becomes the following: each player simultaneously chooses a number of Sybils s_i. Then, A participation games are chosen uniformly at random, and player i enters s_i Sybils in each (note that it is without loss of generality that each player chooses the same number of Sybils per game by symmetry). If A/T' (T' includes the dummy contracts) is small, then even if one player introduces many Sybils, there will still be a decent chance of winding up in a contract where they don't participate at all, which will still yield reasonable reward. However, we certainly need $T' > A$ in order to accomplish this, and the dummy transactions require payment as well.

The plots below describe the following: Assume an initial ratio of A/T (called 'A' in the plots - one can alternatively think of T as being normalized to 1). Then, pick a ratio of dummy contracts to increase T to $T' > A$, and a reward function $f(\cdot)$ of the form $f(m) = c \cdot 2^{-m}$. Player 1 will then pick s_1 to enter in A participation games per round, knowing that all other players will best respond to this, in order to maximize their own payoff. Finally for a given k (desired number of distinct participators per contract), we optimize over all choices of T', c to find the minimum cost solution that guarantees k distinct participators per contract in expectation (in the above form of equilibrium). We include two plots below.

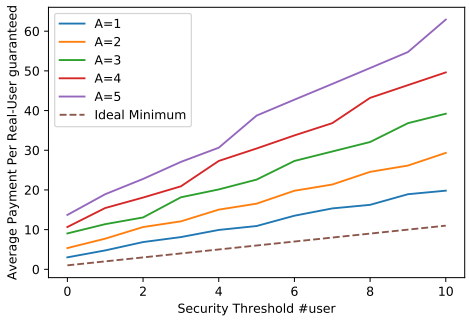

Figure 3: **Plot of total required cost to guarantee x distinct participators in expectation, when one user does optimal Sybil attacks for various initial ratio of A/T.**

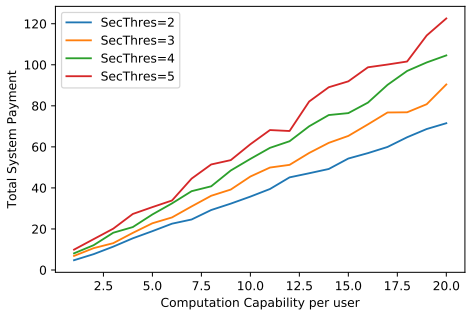

Figure 4: **Plot of total required cost to guarantee $\{2,3,4,5\}$ distinct participators in expectation, when one user does optimal Sybil attacks as a function of initial ratio A/T.**

Both figures have the total cost on the y-axis. Figure 3 has the desired number of distinct participators on the x-axis. The dotted line plots the ideal cost: how much we have to pay per contract to get x distinct verifiers (this is just x). The solid lines plot the cost of the optimal solution using dummy contracts for various initial values of A/T. The takeaway from the first plot is just that there's a noticeable separation between ideal and the necessary cost if $A > T$.

Figure 4 has A on the y-axis, and the solid lines plot the cost of the optimal solution using dummy contracts as a function of A. Here, it is easy to see that the cost is linear in A for all desired number of distinct verifiers. Note that this blowup will come *on top* of whatever blowups are already identified in works based on participation games due to other concerns.