Obfuscated Consensus

(Extended Abstract)

James Aspnes
Yale University
New Haven, United States
james.aspnes@yale.edu

Shlomi Dolev
Ben-Gurion University of the Negev
Beer Sheva, Israel
dolev@cs.bgu.ac.il

Amit Hendin
Ben-Gurion University of the Negev
Beer Sheva, Israel
hendina@post.bgu.ac.il

Abstract

The classic Fischer, Lynch, and Paterson impossibility proof [7, 12] demonstrates that any deterministic protocol for consensus in either a message-passing or shared-memory system must violate at least one of termination, validity, or agreement in some execution. But it does not provide an efficient procedure to find such a bad execution.

We show that for wait-free shared memory consensus, given a protocol in which each process performs at most s steps computed with total time complexity at most t, there exists an adversary algorithm that takes the process's programs as input and computes within $O(st)$ time a schedule that violates agreement. We argue that this bound is tight assuming the random oracle hypothesis: there exists a deterministic **obfuscated consensus protocol** that forces the adversary to spend $\Omega(st)$ time to find a bad execution despite having full access to all information available to the protocol.

This bound is based on a general reduction from constructing an obfuscated consensus protocol to constructing an **obfuscated threshold function** that provably costs $\Omega(t)$ time to evaluate on a single input, where t is a tunable parameter, and for which an adversary with access to the threshold function implementation cannot extract the threshold any faster than by doing binary search. We give a particular implementation of such an obfuscated threshold function that is not very efficient but that is provably secure assuming the random oracle hypothesis. Since our obfuscated consensus protocol does not depend on the specific details of this construction, it may be possible to replace it with one that is more efficient or requires weaker cryptographic assumptions, a task we leave for future work.

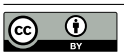
CCS Concepts

• **Theory of computation** → **Distributed algorithms**; **Cryptographic primitives**.

Keywords

asynchronous consensus, distributed algorithms, shared memory, obfuscation, random oracle, threshold functions

ACM Reference Format:
James Aspnes, Shlomi Dolev, and Amit Hendin. 2026. Obfuscated Consensus: (Extended Abstract). In *Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems (ApPLIED'26), July 06–10, 2026, Egham, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3820355.3820383



ApPLIED'26, Egham, United Kingdom

ACM ISBN 979-8-4007-2462-6/2026/07
https://doi.org/10.1145/3820355.3820383

1 Introduction

Asynchronous consensus is one of the most investigated tasks in distributed computing. Consensus over distributed inputs allows a reduction to centralized computation, abstracting away the inherent coordination difficulties that distributed systems cope with.

Asynchronous consensus is a building block in structuring replicated state machines and efficient proof of authority/stake Blockchains, e.g., [5].

Unfortunately, the celebrated result of Fischer, Lynch, and Paterson [7] proves that there is no deterministic algorithm that can yield an asynchronous consensus. Subsequent consensus protocols have escaped this bound by adjusting the requirements of the problem. Paxos [11] succeeded in ensuring the consistency (all participants decide on the same value) and validity (the decided value appears in at least one of the inputs) of the distributed decision but sacrifices the liveness as an adversarial scheduler can keep the system undecided forever. Alternatively, randomization (for example, [4]) allows consensus to be solved at the cost of replacing a deterministic termination guarantee with a probabilistic one, where the protocol finishes with probability 1 in the limit. In these protocols, the adversary scheduler that could otherwise prevent agreement following the procedure given by the FLP proof is unable to predict the future and thus falls off the bad path.

We investigate an alternative approach, where validity and termination are guaranteed in all executions, and agreement holds for all but a small number of executions that are computationally expensive to compute, even for an adversary given access to the entire code and knowledge of the processes in the protocol. This idea is inspired by the Fiat-Shamir heuristic [6] for extracting a zero-knowledge challenge from a hash of public values assuming a random oracle. In our approach, the scheduling choices of the adversary in an asynchronous shared-memory system, as observed by the processes, become the input to an obfuscated threshold function. Each process computes its output by computing a single value of this function, while the adversary can only prevent agreement through the much more expensive process of finding the threshold input at which the value of the function changes from 0 to 1. This gap can in principle be exploited to obtain a fully deterministic wait-free asynchronous protocol that solves consensus in most executions, assuming the adversary has limited computational power.

This approach has some limitations. We show that, given any protocol in which each process performs s steps and computes for

t time, there is an adversary program that outputs an agreement-violating schedule in $O(st)$ time in the worst case.

Conversely, we provide a reduction from our obfuscated consensus problem to constructing an obfuscated threshold function, and give a particular implementation of such a function as an obfuscated truth table. Here each bit of the truth table is hidden using a post-quantum time lock puzzle (see, e.g., [1, 15] and the references therein) for efficiently structuring obfuscated programs (see, e.g., [9] and the references therein). This allows us to tune the base cost of computing the threshold function for both the processes and the adversary, and to prevent the adversary from finding the hidden threshold any faster than by binary search over a range exponential in the step complexity of the consensus protocol.

While our particular implementation of an obfuscated threshold function involves a relatively expensive setup phase, it is possible to imagine an implementation using more sophisticated cryptographic techniques that would avoid this overhead. We discuss this possibility further in Section 7.

Before moving on, we would like to discuss why investigating such adversarial settings is interesting in the first place. In the asynchronous setting, the purpose of the adversarial scheduler is to theoretically induce the worst-case scenario on the protocol; if a protocol works under such settings, then no schedule can cause it to violate its correctness conditions or loop indefinitely. In randomized protocols, theoretically, the adversary cannot guess the next configuration with certainty and thus cannot know the scheduling that will keep the system in a bad state. Solely based on the theoretical implications the motivation for a computationally constraint adversary naturally arises: if we can hide the next configuration behind a computational puzzle that is too difficult for the adversary, it cannot not find the bad schedule with certainty. The real question arises when looking at the practical implications of this. For protocols that guarantee agreement and validity, bad schedules are infinite loops, timing messages or scheduling processes in just the right order such that the system forever remains in an indecisive state. In the randomized case, if the protocol enters a loop of indecisive configurations, then the randomized step can pull it out by randomly moving the system to a different configuration and breaking the loop. The question remains: what is the practical implication of the computationally bounded adversary? Resilience against an actual adversary. Rather than attempt to pull the protocol out of non-terminating loops, we defend the protocol against real scheduling attacks such as Adversarial Transaction Reordering, that occur when miners, validators, or automated bots manipulate the sequence of pending transactions in a block to extract financial profit. By exploiting the time delay between when a transaction is broadcast and when it is confirmed, bad actors engage in front-running, back-running, and sandwich attacks at the expense of regular users. By obfuscating the keys parts of the protocol behind a cryptographic puzzle, we can create a time delta between the honest participants and the malicious ones, preventing adversarial attacks on the schedule in cases where the adversary algorithm cannot solve the cryptographic puzzle fast enough.

Connection to blockchains. Our scheduler model abstracts schedule attacks that arise in blockchain systems when an adversary influences the order in which transactions, messages, or block proposals are observed and processed. In this correspondence, processes represent validators or protocol participants, shared memory represents the publicly evolving protocol state, and a schedule represents the adversary's chosen ordering of protocol events. A bad asynchronous schedule therefore corresponds to a strategically chosen ordering that drives the system into an undesirable execution, such as disagreement, delayed decision, or exploitable transaction reordering. Obfuscated consensus captures the goal of making such an ordering computationally expensive to find: honest participants execute the protocol directly, while the adversary must search for the damaging schedule.

1.1 Overview

The system settings are described in the next section. The problem of **obfuscated consensus** is described in Section 3: this replaces the usual agreement condition for consensus that requires all processes output the same value in each execution with a new **obfuscated agreement** condition that only requires that it is computationally expensive for the adversary scheduler to find a schedule that violates agreement. Section 5 gives a reduction from obfuscated consensus to a purely cryptographic problem, that of obfuscating a threshold function. This reduction has parameters allowing both the computation cost incurred by each process and the ratio of the adversary's cost to each process's cost to be adjusted independently. An implementation of an obfuscated threshold function, along with a proof of security, is given in Section 6. Finally, we discuss some of the limitations of our current implementation and possibilities for future work in Section 7.

Proofs are omitted from this extended abstract and can be found in [2].

2 Model

We use a standard asynchronous shared-memory model, with some small tweaks to include explicitly the programs used by the processes and adversary to compute their choices.

The system consists of n asynchronous **processes** $p_1, ..., p_n$ that communicate by reading and writing to shared objects, with timing controlled by an **adversary scheduler**.

The behaviour of each process is controlled by a random-access machine program P that implements a function $Q \times V \to Q \times A$, where Q is the set of states of the process, V is the set of possible return values of operations (including a special null value used to indicate the first step by the process), and A is the next operation to execute. We consider the cost of computing the transition function a single time to be constant.

The adversary scheduler is a program in the same model that takes as input a step bound s, a sequence of programs $\langle P_1, \ldots, P_n \rangle$ and initial states $q_1, \ldots, q_n$, and outputs a sequence of process ids, where each id occurs at most s times, specifying the order in which the processes take steps. We will refer to this sequence of process ids as a **schedule**.

Communication is via **shared memory**: a collection of **atomic read/write registers** $r_1, \ldots, r_m$, where applying a `read` operation to a register returns its current value and applying a `write` operation sets the value and returns nothing.

A **configuration** of the system consists of the states of the processes and registers, together with at most one pending operation for each process.

Every schedule yields an **execution**, an alternating sequence of configurations and operations $C_0\alpha_1, C_1\alpha_2C_2\ldots$ in which each C_{i+1} is the result of applying operation α_{i+1} to configuration C_i. Because the processes are deterministic, the adversary can simulate the execution corresponding to any particular schedule in time proportional to the total time used by the processes to choose their steps in that execution.

We assume that both the processes and the adversary have access to a **random oracle** H, which is an idealized version of the cryptographic hash function. The random oracle acts as a black box that can be queried with input x to return output y. The box is consistent; that is, any input x will always output the same y. The box is uniformly random, which means the probability of getting output y for an input x that was not already queried is uniform. Though everyone has access to this black box and can query it, its internal workings are unknown (see Chapter 5 of [10]).

3 Obfuscated consensus

A consensus protocol is a protocol that is run by a set of n asynchronous processes, each of which starts with a binary input value $v_i \in \{0,1\}$, runs until it reaches a decision value $d_i \in \{0,1\}$, then halts. A consensus protocol is correct if it satisfies the following requirements:

Agreement A consensus protocol satisfies agreement if no two processes decide on different values: $\forall i, j : d_i = d_j$.

Validity A consensus protocol is valid if all processes decide on a value that was the input of some process, $\forall i \exists j : d_i = v_j$.

Termination Every non-faulty process decides after a finite number of steps.

We think of the consensus problem as an adversarial game between the scheduler and the processes. The processes try to reach agreement on the same value by reading and writing to shared objects, and the scheduler tries to prevent them from doing so by strategically ordering these operations.

The Fischer-Lynch-Paterson (FLP) impossibility result [7], as extended to shared memory by Loui and Abu-Amara [12], shows that in an asynchronous system with $n \geq 2$ processes in which at least one process can fail by crashing, there is always an adversary strategy that finds an execution in which at least one of the three conditions is violated. We will be interested in constructing protocols where finding this strategy is computationally difficult.

Define an **obfuscated consensus protocol** to be a protocol that satisfies termination and validity always, but replaces agreement with a condition that we can describe informally as

Obfuscated agreement The cost to the adversary to compute a schedule that violates agreement substantially exceeds the cost to the processes to execute the protocol.

More formally, the adversary must solve an **obfuscated consensus problem**: Given process programs $P_1, \ldots, P_n$ and a step bound s, find a schedule in which each process takes s steps that violates one of termination, validity, or agreement. A successful obfuscated consensus protocol is one for which solving this problem is expensive, for some reasonable definition of expensive. We will start by bounding this cost from above.

4 Bounding the cost of finding a bad execution

The core idea of the FLP impossibility proof [7] is the notion of **bivalence**. A configuration C is bivalent if there is a path from it to a configuration where a process decides on the value 1, and there is also a path from it to another configuration where a process decides on the value 0. Similarly, a configuration C is univalent if all the paths from C lead to configurations with an identical decision value. Because of the agreement property, any configuration in which some process has decided is necessarily univalent.

The FLP proof, in both its original form [7], and its extension to shared memory by Loui and Abu-Amara [12], gives an explicit procedure for constructing an infinite non-terminating execution for any protocol that satisfies agreement and validity, by starting in a bivalent configuration and always choosing bivalent successor configurations. However, the computational cost of this construction is not considered. We would like to find an adversary strategy that allows it to prevent a protocol from terminating without having to explore the space of all executions.

The time complexity of computing a non-terminating, and thus infinite, execution is necessarily infinite. We will instead look at the more tractable problem of computing an extension for some fixed number of steps s, which is the justification for defining obfuscated consensus as satisfying termination and validity always but possibly failing agreement. As described in Section 2, we replace an abstract adversary function with a concrete adversary program that takes as input the starting configuration of the system and the programs used by the processes to choose their next operations and, ultimately, their outputs.

Formally, let us define the following problem. We assume that each program P_i has two special return actions R_0 and R_1, that it must choose between after exactly s steps; these provide the output of each process in the consensus protocol. The adversary, a program that takes as input the programs $P_1, \ldots, P_n$ and the step bound s and constructs a schedule in which each process takes exactly s steps, wins if any of the processes' outputs disagree.

The following theorem shows that the cost to the adversary to find a bad execution, relative to the worst-case cost of each process to carry out some execution, scales linearly with the number of steps performed by each process.

Theorem 4.1. *Given inputs $P_1, \ldots, P_n$ representing processes in a shared-memory system where, in all executions, (a) each process p_i outputs a decision value in s steps, and (b) each process p_i uses a total of at most t time to compute its transitions, there is an adversary program that computes in $O(st)$ time a schedule that causes some pair of processes to output different decision values.*

To prove the theorem, we consider a restricted version of the FLP bivalence argument in which valence is defined only in terms of solo extensions of the current configuration. This will significantly reduce the number of extensions the adversary needs to consider.

For each configuration C, define the **preference** $\texttt{pref}_p(C)$ of p in C to be the output value of p in the execution $C\alpha$ where only p takes steps before it decides. We will call such an extension α p's

solo-terminating extension and write $\texttt{pref}_p(C) = \texttt{decision}_p(C\alpha)$ to indicate that p decides this value in configuration $C\alpha$. We know that a unique such extension α exists because p is deterministic (making α unique) and the system is wait-free (so that no fairness requirement prevents p from running alone).

Following Gelashvili [8], call a configuration C **solo-bivalent** if there are processes p and q such that $\texttt{pref}_p(C) \neq \texttt{pref}_q(C)$. Call a configuration **solo-b-valent** if $\texttt{pref}_p(C) = b$ for all p. Similar to the original FLP construction, our goal is to start in a solo-bivalent configuration and stay in a solo-bivalent configuration.

We start with some simple observations about how a process's preference can change. Given executions α and β, write $\alpha \sim_p \beta$ (α is **indistinguishable** by p from β) if $\alpha|p = \beta|p$, meaning that p observes the same events in both executions. Note that $\alpha \sim_p \beta$ in particular implies $\texttt{decision}_p(\alpha) = \texttt{decision}_p(\beta)$.

Lemma 4.2. *Let C be a configuration and let π be an operation of p. Then $\texttt{pref}_p(C\pi) = \texttt{pref}_p(C)$.*

Lemma 4.3. *Let π be a read operation by p. Then $\texttt{pref}_q(C\pi) = \texttt{pref}_q(C)$.*

Lemma 4.4. *Let π_p and π_q be writes by p and q to the same register. Then $\texttt{pref}_p(C\pi_q\pi_p) = \texttt{pref}_p(C\pi_p)$.*

Lemma 4.5. *For any wait-free shared-memory protocol satisfying termination and validity, there exists an initial solo-bivalent configuration for any $n \geq 2$.*

Lemma 4.6. *Let C be a solo-bivalent configuration where p and q are processes with $\texttt{pref}_p(C) \neq \texttt{pref}_q(C)$. Let π_p and π_q be the pending operations of p and q in C. Then, at least one of the following holds:*

1. *$C\pi_p$ is solo-bivalent with $\texttt{pref}_p(C\pi_p) = \texttt{pref}_p(C)$ and $\texttt{pref}_q(C\pi_p) = \texttt{pref}_q(C)$.*
2. *$C\pi_q$ is solo-bivalent with $\texttt{pref}_p(C\pi_q) = \texttt{pref}_p(C)$ and $\texttt{pref}_q(C\pi_q) = \texttt{pref}_q(C)$.*
3. *$C\pi_p\pi_q$ is solo-bivalent with $\texttt{pref}_p(C\pi_p\pi_q) \neq \texttt{pref}_p(C)$ and $\texttt{pref}_q(C\pi_p\pi_q) \neq \texttt{pref}_q(C)$.*

One difference between solo-bivalence and bivalence that is illustrated by the third case of Lemma 4.6 is that it is not necessarily the case that a configuration with a solo-bivalent successor is itself solo-bivalent. But we are happy as long as we can find solo-bivalent extensions of solo-bivalent configurations, even if this means passing through intermediate configurations that may not be solo-bivalent.

To turn Lemma 4.6 into an algorithm, we just need to be able to test which of its three cases holds. To compute a bad schedule, the adversary carries out the following steps:

1. Pick two processes p and q, and start in a configuration C where $\texttt{pref}_p(C) = 0$ and $\texttt{pref}_q(C) = 1$, as in Lemma 4.5.
2. Given C with $\texttt{pref}_p(C) \neq \texttt{pref}_q(C)$:
   (a) If $\texttt{pref}_q(C\pi_p) = \texttt{pref}_q(C)$, append π_p to the schedule and set $C \leftarrow C\pi_p$.
   (b) If $\texttt{pref}_p(C\pi_q) = \texttt{pref}_p(C)$, append π_q to the schedule and set $C \leftarrow C\pi_q$.
   (c) Otherwise, the third case of Lemma 4.6 holds. Append $\pi_p\pi_q$ to the schedule and set $C \leftarrow C\pi_p\pi_q$.
3. Repeat until one of p and q decides; then run the other to completion, appending its operations to the schedule.

The cost of computing $\texttt{pref}_p(C)$ for any configuration C is $O(t)$, since we can just simulate p until it decides. To avoid copying, we do this simulation in place, logging any changes to p's state or the state of the shared memory so that we can undo them in $O(t)$ time. For computing $\texttt{pref}_p(C\pi_q)$, we again pay at most $O(t)$ time, since applying π_q to C takes no more than $O(t)$ time, as does running p to completion thereafter. The same bounds hold with the roles of p and q reversed.

It follows that each iteration of the main body of the algorithm finishes in $O(t)$ time. Since each of p and q do at most s steps before deciding, there are at most $2s = O(s)$ iterations, giving a total cost of $O(st)$, even taking into account the $O(t)$ cost of the last phase of the algorithm. Since every iteration of the algorithm yields a solo-bivalent configuration, the preferences of p and q are never equal, and so the constructed schedule violates agreement. This concludes the proof of Theorem 4.1.

5 Obfuscated consensus from obfuscated threshold

The ratio between the $O(t)$ time complexity of each process in Theorem 4.1 and the $O(st)$ time complexity of the adversary suggests that a useful strategy for the processes faced with a computationally-limited adversary is to exhaust the adversary's resources by forcing it to simulate a large number of expensive potential executions. We can do this by making the outcome of each process's execution depend on evaluating an expensive function of the view that process has in the execution.

An **obfuscated threshold function** $f : \{0, \ldots, \ell\} \to \{0, 1\}$ is a function, represented as a random-access machine program, with $f(0) = 0$ and $f(\ell) = 1$. The **obfuscated threshold problem** is to find, given a representation of such a function, an input v such that $f(v) \neq f(v+1)$. We will show that finding a bad execution of a particular length of any obfuscated consensus protocol is equivalent to solving this problem.

Binary search will find a v with $f(v) \neq f(v+1)$ in $O(\log \ell)$ evaluations of f, which puts an upper bound on the cost of solving the obfuscated threshold function problem. But it may be that a sufficiently clever algorithm can extract v from f without evaluating f explicitly. We will discuss how to build an f that resists such attacks in Section 6.3.

Note that the end conditions $f(0) = 0$ and $f(\ell) = 1$ show that at least one transitional value v always exists, and, for some choices of f, many such transitional values exist. This makes the name "obfuscated threshold function" a bit misleading, since f is not necessarily a threshold function. But choosing a threshold function in particular reduces the likelihood of guessing a bad v at random, so we will try to produce threshold functions if we can.

We will show a reduction between these problems by constructing an obfuscated consensus protocol that uses an obfuscated threshold function to compute its output values. In Algorithm 5.1, we give a protocol that assumes that every process (and thus also the adversary) has access to an obfuscated threshold function f, and show that the adversary computes a bad execution for this protocol if and only if it can find a v such that $f(v) \neq f(v+1)$.

```
shared data: array A[1...s][2] of atomic registers,
             initialized to ⊥
procedure approximateAgreement(input, s)
    i ← input
    for r ← 1...s do
        // write my current position
        A[r][i mod 2] ← i
        // read opposite position
        i' ← A[r][(i + 1) mod 2]
        if i' = ⊥ then
            // no opposition, stay put
            i ← 2 · i
        else
            // adopt midpoint
            i ← i + i'
    return i
procedure obfuscatedConsensus(input, f, s)
    i ← approximateAgreement(input, s)
    return f(i)
```

Algorithm 5.1: Obfuscated consensus based on an obfuscated threshold

Pseudocode for a protocol that does this is given in Algorithm 5.1. The main procedure `obfuscatedConsensus` takes as arguments a consensus input value input $\in \{0, 1\}$, a representation of a function f that converts integer values to Boolean decisions, and a security parameter s that scales the number of shared-memory steps taken by the process.

The protocol proceeds in two phases.

First, the processes carry out a s-round approximate agreement protocol that assigns each process p a value v_p in the range $0 \dots 2^s$, with the properties that

(1) Any process that sees only input v obtains value $2^s \cdot v$.
(2) For any two processes p and q, $|v_p - v_q| \leq 1$.

The approximate agreement protocol is an adaptation of Moran's one-dimensional midpoint protocol [13]. In Moran's original protocol, each process takes a sequence of snapshots of current values and adopts the midpoint of the values it sees until all fit within a particular range. In our protocol, we organize this sequence into a layered execution of s rounds, where in each round, a process adopts the average value it sees, scaled by a factor of 2 per round to track the integer numerators instead of the actual fractional values. By taking advantage of the inputs starting at 0 and 1, we can show that at most two values, differing by at most 1, appear at the end of each round; this allows us to replace the snapshot with a pair of registers, one of which holds the even value and one the odd. Formally:

Lemma 5.1. *Let S_r be the set of all i values that are held by any process after r iterations of the loop in procedure* `approximateAgreement`. *Then $S_r \subseteq \{v_r, v_r + 1\}$ for some $0 \leq v_r \leq 2^r - 1$.*

In particular, the set of output values is given by $S_s \subseteq \{v_s, v_s + 1\}$ where $0 \leq v_s \leq 2^s - 1$.

To obtain its decision value, each process p then feeds its output v_p to a function $f(v_p)$. To ensure validity, we require that $f(0) = 0$ and $f(2^s) = 1$. Agreement is obtained if $f(v_p) = f(v_q)$ for all processes p and q.

If f requires T time to evaluate, the time complexity incurred by each process running Algorithm 5.1 is $O(s) + T$. From Theorem 4.1 this implies that there is an adversary strategy that computes a bad execution in time $O(s^2 + sT)$; this also tells us that at least one bad execution exists. We'd like to show a comparable lower bound on the cost to the adversary, assuming it is difficult to find a threshold in f.

Since every process has a value in $\{v_s, v_s + 1\}$, agreement holds automatically if $f(v_s) = f(v_s + 1)$. So an adversary that finds an execution that produces disagreement also finds a value v_s with $f(v_s) \neq f(v_s + 1)$.

Thus, restricting the adversary and processes to have only oracle access to f induces the cost separation, where each process computes f only once, and the adversary has to search for the threshold.

For any fixed f with $f(0) = 0$ and $f(2^s) = 1$, there is a random-access machine that outputs a threshold value v with $f(v) \neq f(v+1)$ in $O(1)$ time, since it can just include v in its code. We will need to avoid this by considering the average-case cost to find a threshold value for an f drawn from some family of threshold functions. Applying Algorithm 5.1 to the functions in this family then give a family of consensus protocols.

Theorem 5.2. *Let F be a family of threshold functions with range $\{0, \dots, \ell\}$. Fix some n and some s such that $2^s \geq \ell$, and for each $f \in F$, let P_f be the obfuscated consensus protocol given by instantiating the s-round version of Algorithm 5.1 with f as its decision function.*

Suppose we sample P_f uniformly at random and consider a starting configuration in which not all processes have the same input. If there is a probabilistic random-access machine M that takes P_f and the process's initial states as input, runs for T expected time, and outputs a schedule that produces disagreement, then there is a probabilistic random access machine M' that takes f as input, runs for $T + O(ns)$ expected time, and outputs a v with $f(v) \neq f(v + 1)$.

We clarify that the sampling is used only to state average-case hardness over protocol instances. After f is sampled, the protocol P_f is deterministic. This prevents the adversary from knowing the hidden threshold a head of time from a previous execution of the protocol.

In the following section, we give an implementation of a family of threshold functions with these properties.

6 Implementing an obfuscated threshold function

Because the adversary can observe the processes' programs, we cannot enforce oracle-only access to f. We therefore need an obfuscated implementation of f that is expensive to evaluate and reveals, when evaluated at v, no information about $f(v')$ beyond what follows from the threshold-function property.

6.1 Definition

We introduce a new approach for computationally obfuscating programs that use one-way function primitives. The program's creation uses randomization that can be revealed by (tunable) inversion processing required to invert the one-way function. We construct the obfuscation by using a random oracle to hide a sequence of bits that encode the truth table of a threshold function. To obtain the output from the obfuscation, one must find the pre-image of a single hash. Thus it is imperative that the hardness of computing the pre-image be tunable such that it is feasible to find a single pre-image, yet infeasible for the computationally bounded adversary to check enough pre-images to de-obfuscate the program. We achieve this tunability not by weakening the hash function, but by providing a prefix for the pre-image, such that the processes need only complete it. We begin with the following definitions.

Definition 6.1 (Threshold Function)**.** *Let $T, \ell \in \mathbb{N}$ be two integers such that $0 < T < \ell$. Define function $f_T : [\ell] \to \{0,1\}$ as*

$$f_T(i) = \begin{cases} 0 & i < T \\ 1 & i \geq T \end{cases}$$

Then f_T is a threshold function for threshold T.

Definition 6.2 (Random Oracle Hash Function)**.** *Let $m \in \mathbb{N}$ and let $H_m : \{0,1\}^m \to \{0,1\}^m$ be sampled uniformly at random from the set of all functions $\{0,1\}^m \to \{0,1\}^m$ (i.e., H_m is a* random oracle */ random function). *Then for all distinct $x \neq x' \in \{0,1\}^m$ and all $y, y' \in \{0,1\}^m$,*

(1) $\Pr[H_m(x) = y] = 2^{-m}$
(2) $\Pr[H_m(x) = y \ \wedge \ H_m(x') = y'] = 2^{-2m}$

Definition 6.2 is adapted from Katz and Lindell [10], Chapter 5.1. Brute force search needs at most 2^m trials to find a pre-image; the next corollary states that a known pre-image prefix reduces this search space.

Corollary 6.3 (Tunable pre-image runtime)**.** *Let $H_m : \{0,1\}^m \to \{0,1\}^m$ be a random oracle (Definition 6.2). Fix $k < m$ and a prefix $p \in \{0,1\}^k$, and let $H_m(p\|r) = y$ be the output of H_m for some extension r of p. For any (possibly randomized) algorithm $\mathcal{X}$ that makes at most q oracle queries to H_m and outputs $z \in \{0,1\}^{m-k}$, it holds that*

$$\Pr_{H_m, \mathcal{X}}[H_m(p\|z) = y] \leq \frac{q}{2^{m-k}}$$

The parameter k here is used to tune the size of the pre-image search space, by exposing the first k bits of the pre-image later in the obfuscation construction.

Definition 6.4 (Threshold obfuscation scheme)**.** *Let $\Pi = (Prep, Probe)$ be a pair of algorithms. Given a representation of a threshold function f_T (6.1), if $Prep$ produces output $O_T = Prep(f_T)$ such that, for any given input x of f_T, $Probe(O_T, x) = f_T(x)$ holds with high probability, then Π is a threshold obfuscation scheme.*

We move one to our security definition. We base this definition on the adversarial experiment format used in the book by Katz & Lindell [10]. There exists a definition for obfuscation in [3] (Boaz Barak et al.). This definition requires that no probabilistic polynomial-time adversary can generate the same outputs as the original program with high probability. While we would like to have such obfuscation for the threshold function, our current obfuscation construction does not satisfy this very strong notion; rather, we require that to reproduce the same outputs, an adversary must have a runtime that is at least logarithmic in the size of the construction, and an adversary must reproduce the exact outputs with probability 1; otherwise, it cannot do any better than a random guess. Specifically, since we tailor this obfuscation to a threshold function, we formulate this specific notion of obfuscation as an experiment where the adversary gets the obfuscated threshold function just like the processes, and must recover the threshold T in order to succeed.

Definition 6.5 (Obfuscation Security Experiment $Obf_{\mathcal{A},\Pi,\ell}$)**.** *Let $\ell \in \mathbb{N}$ be the range of values for threshold T. Let $\Pi = (Prep, Probe)$ be an obfuscation scheme (6.4). Let $\mathcal{T}_{Probe}$ be the runtime of algorithm $Probe$. Let $\mathcal{A}$ be an adversarial algorithm with runtime $\mathcal{T}_{\mathcal{A}} < \mathcal{T}_{Probe} \cdot \log \ell$.*

(1) *A random $T \in [\ell]$ is chosen.*
(2) *Compute $O_T = Prep(f_T)$.*
(3) *$\mathcal{A}$ is given O_T.*
(4) *$\mathcal{A}$ returns an integer T'.*
(5) *Let τ be the number of times $\mathcal{A}$ computes H_m.*
(6) *The output of the experiment is $1, \tau$, if $T' = T$, otherwise the output is $0, \tau$.*

Using Definition 6.5, we formulate a definition of security for a threshold obfuscation scheme. Defining the security as an upper bound on the probability of success in the experiment Obf would yield a strong definition. However, in our case, since the adversary must be able to cause the protocol to terminate with certainty, we can relax the definition and require that the expected number of hashes the adversary must compute in order to find T is bounded. This approach allows us to simplify the security proof.

Definition 6.6. *A threshold obfuscation scheme (Definition 6.4) $\Pi = (Prep, Probe)$ is secure if for every adversary algorithm $\mathcal{A}$, it holds that*

$$\mathbb{E}[\tau \mid b = 1] \geq \frac{1}{2}\left((2^{m-k} + 1)\log(\ell + 1)\right)$$

where $b, \tau = Obf_{\mathcal{A},\Pi,\ell}$.

Note that increasing k exposes a greater portion of the pre-image, shrinking the search space, and reducing the expected number of probes required to find T. While a bound on the probability of success would be stronger, this definition is sufficient to show that the adversary must do more work than the processes in order to cause the protocol to fail. We move on to define our threshold obfuscation scheme. We begin with the $Prep$ algorithm, the output of which will be used to compute the output of the threshold function in the $Probe$ algorithm. We call this algorithm preprocess.

preprocess

(1) Fix integers $1 < k < m$ and $\ell \geq 2$.
(2) Choose threshold $1 < T < \ell$ uniformly at random.
(3) For all $1 \leq i \leq \ell$, sample binary strings $P_i \leftarrow \{0,1\}^k$ and $r_i \leftarrow \{0,1\}^{m-k}$ uniformly at random.

(4) For all $1 \le i \le \ell$
  (a) compute $d = H(P_i \| r_i)$
  (b) set $C_i \leftarrow d_1 d_2 \dots d_{m-1}$ that is, C_i is the first $m-1$ bits of d
  (c) set $v_i \leftarrow d_m \oplus f_T(i)$
(5) Return $(P_1, \dots, P_\ell, C_1, \dots, C_\ell, v_1, \dots, v_\ell)$.

Next, we define the *Probe* algorithm, which we will use to compute a single output of f_T.

probe

(1) Assume access to H_m as well as $C_1, \dots, C_\ell, P_1, \dots, P_\ell$, and $v_1, \dots, v_\ell$.
(2) Let $1 \le i \le \ell$ be an input of f_T.
(3) Find string $r \in \{0,1\}^{m-k}$ such that $[H_m(P_i \| r)]_j = [C_i]_j$ for all $j = 1, \dots, m-1$.
(4) Return $[H_m(P_i \| r)]_m \oplus v_i$

A more complete definition and implementation can be found in Section 6.3.

6.2 Correctness & security

We show that Algorithms 6.1 and 6.2, respectively `preprocess` and `probe`, form a threshold obfuscation scheme (6.4) satisfying Definition 6.6: Claim 6.8 proves correctness and Claim 6.10 proves security. We begin with a lemma.

Lemma 6.7. *[Distinct pre-image with prefix] Let $k, m \in \mathbb{N}$ be two integers where $k < m$, and let H_m be a random oracle 6.2. Let $p \in \{0,1\}^k$, $r \in \{0,1\}^{m-k}$ be two binary strings. It holds that*

$$\Pr\left[\exists_{x \neq r} (H_m(p||x) = H_m(p||r))\right] \le 2^{-k}$$

The following is the construction correctness claim. We require that the parameters k, ℓ satisfy $\frac{\ell}{2^k} = negl(k)$, meaning that their ratio is negligible in k. This assumption is reasonable since increasing k decreases the probability of a collision and reduces the amount of work required to find the suffix of a pre-image.

Claim 6.8. *The algorithm pair* (`preprocess`, `probe`) *is a threshold obfuscation scheme 6.4, for ℓ, k that satisfy $\frac{\ell}{2^k} = negl(k)$.*

Lemma 6.9. *For any adversary $\mathcal{A}$ that recovers T from C, P, V, it holds that*

$$\mathbb{E}[\tau] \ge \frac{1}{2}\left((2^{m-k}+1) \cdot \log(\ell+1)\right)$$

where τ is the number of times $\mathcal{A}$ computes H_m.

The following is a security claim based on the established definition.

Claim 6.10. *The threshold function obfuscation* Π = `preprocess`, `probe` *is secure as defined in 6.6.*

6.3 Implementation

Here we give a more detailed description of the obfuscated threshold implementation outlined in Section 6.1.

```
1  procedure preprocess(ℓ, T, m, k s.t. t ≤ ℓ ∧ k < m)
       // Initialize array of hashes and array of
          nonces
2      C[1, ..., ℓ][1, ...m − 1] an uninitialized array of bit strings
3      P[1, ..., ℓ][1, ...k] an uninitialized array of bit strings
4      V[1, ..., ℓ] an uninitialized array of bits
5      for i ← 1 ... ℓ do
6          P[i] ← random bit string of length k
7          r_i ← random bit string of length m − k
8          d ← H_m(P[i]‖r_i)
9          C[i] ← d_1 d_2 ... d_{m−1}
10         V[i] ← d_m ⊕ f_T(i)
11     return C, P, V
```

Algorithm 6.1: Threshold Encapsulation

6.3.1 Preprocessing Algorithm. We present an implementation of the first algorithm of our threshold obfuscation scheme, Algorithm 6.1. This algorithm accepts hash size parameter m, hardness parameter k, value range ℓ, and input threshold T. It returns three arrays, C, P and V. The idea is to hide the threshold T in an array of bits $V = v_1, \dots, v_\ell$. We create an array of hashes $C = C_1, \dots, C_\ell$ where each hash is created by randomly choosing two strings $p_i \in \{0,1\}^k$ and $r_i \in \{0,1\}^{m-k}$ for each $i = 1, \dots, \ell$ and hashing their concatenation $p_i \| r_i$. We store in C_i the first $m-1$ bits of the hash $H_m(p_i \| r_i)$ and the last bit we XOR with the value of $f_T(i)$ to obscure it. We store this in the bit $v_i \leftarrow [H_m(p_i \| r_i)]_m \oplus f_T(i)$. In this way, in order to compute $f_T(i)$, one must find a string y such that $[H_m(p_i \| y)]_{1\dots(m-1)} = C_i$, and then compute $[H_m(p_i \| y)]_m \oplus v_i$. Given that there is a high probability that $[H_m(p_i \| y)]_{1,\dots,(m-1)} = [H_m(p_i \| r_i)]_{1,\dots,(m-1)} \implies [H_m(p_i \| y)]_m = [H_m(p_i \| r_i)]_m$ it follows that, $[H_m(p_i \| y)]_m \oplus v_i = [H_m(p_i \| y)]_m \oplus [H_m(p_i \| r_i)]_m \oplus f_T(i) = f_T(i)$ holds w.h.p. To create the hashes, we employ our random oracle function (6.2). Since one needs to find a suffix for a pre-image of the hash (up to the last bit) in order to find $f_T(i)$, we must ensure that finding such a suffix is feasible. To control the size of the search space we have a security parameter k which determines the size of the prefix string p_i. For this reason, the preprocessing algorithm returns the array of prefix strings $P = P_1, \dots, P_\ell$. Increasing k decreases the size of the suffix required to find, which automatically reduces the search space. By the properties of a random oracle (6.2), reducing the size of the search space implies reducing the runtime required to find r_i.

In a practical implementation of the construction, the standard hash [14] SHA 512/256 can be used in place of H_m (Definition 6.2). While no formal proof exists, it is widely accepted that in practice, SHA behaves like a random oracle, including an avalanche effect, where any minor change in the input, even a single bit, drastically alters the output.

6.3.2 Outline. Algorithm 6.1 is executed by the programmer to generate an obfuscated protocol. The algorithm begins by initializing the commitment array C, nonce array P, and verification-bit array V (lines 1 to 3). It then iterates over all ℓ entries (lines 4 to 10). For each index i, the algorithm generates a random k-bit nonce $P[i]$ (line 5) and a random $(m-k)$-bit string r_i (line 6). It

computes $d = H_m(P[i]\|r_i)$ (line 7), stores the first $m-1$ bits of d in $C[i]$ (line 8), and stores the last hash bit masked with $f_T(i)$ in $V[i]$ (line 9). Finally, the algorithm returns the arrays C, P, and V (line 11).

local data: integers ℓ, m, k, precomputed array of hashes $C[1, ..., \ell][1, ..., m-1]$, prefixes $P[1, ..., \ell][1, ..., k]$, and bits $V[1, \ldots, \ell]$

1 **procedure** probe($i \in [\ell]$)
2 $Y[1, ..., m-k]$ an uninitialized bit string
3 $D[1, ..., m]$ an uninitialized bit string
4 $Y \leftarrow$ string that satisfies $C[i][1, \ldots, m-1] = H_m(P[i]\|Y)$
5 $D \leftarrow H_m(P[i]\|Y)$
6 **return** $D[m] \oplus V[i]$

Algorithm 6.2: Threshold Probe

6.3.3 Probing algorithm. The processes use the probe procedure described in Algorithm 6.2 to compute $f_T(i)$.

The algorithm works by finding the pre-image $P[i]\|r_i$ that was used to compute $C[i]$ in the preprocessing phase. It begins by initializing a bit string Y of length $m-k$ to hold potential suffixes of the pre-image $P[i]$ of $C[i]$, and an m-length bit string D to hold the hash that is equal to $C[i]$ in the first $m-1$ bits. Next, it finds and stores in Y a suffix for the pre-image of $C[i]$ such that the first $m-1$ bits of the hash of $P[i]\|Y$ are equal to $C[i]$ (line 3). It stores this hash in D (line 4). This means that, with high probability $D[m] = H_m(P[i]\|r_i)[m]$. The exact method of finding this pre-image has not been specified, but exhaustive search mining or any other method to find a pre-image of a cryptographic hash function can be employed. Thus, Algorithm 6.2 returns $f_T(i)$ by computing $D[m] \oplus V[i]$ (line 5), since the preprocessing algorithm defined $V[i] = H_m(P[i]\|r_i)[m] \oplus f_T(i)$, the algorithm returns $f_T(i)$.

7 Conclusion

Designing an algorithm to cope with an adversarial scheduler (and/or inputs) ensures the algorithm's function in rare scenarios where the schedule is the most unfortunate for the algorithm. In many cases, the worst case leads to an impossibility result, while the rare scenario does not happen in practice. This is particularly true in scenarios that require ongoing tracing and computation of the imaginary scheduler entity.

We examine the power of using program obfuscation and random oracles to derandomize distributed asynchronous consensus algorithms. The power of the random oracle can be demonstrated in algorithms where symmetry is broken by other means, such as process identifiers and/or symmetry-breaking operations, such as compare-and-swap. Thus, randomization has an inherent more substantial power in breaking symmetry. On the other hand, harvesting proper randomization during runtime is challenging, and it should be avoided if possible.

A weakness of our construction is that it requires substantial preprocessing to construct an obfuscated threshold function since our implementation is just a truth table hidden behind time-lock puzzles. A natural question is whether a more sophisticated obfuscation procedure could reduce this setup, perhaps by constructing a function $f(s, i)$ where the threshold T is a hash of some shared data s but the implementation of f prevents recovering T more efficiently than simply doing binary search.

However, even with these limitations, we believe that we have demonstrated that integrating program obfuscation and random oracle abstractions and functionalities in distributed computing is helpful for the theory and practice of distributed computing and systems.

Acknowledgments

This work was supported by the Google Research Grant, the Rita Altura Trust Chair in Computer Science, the Frankel Center for Computer Science, and the Israeli Science Foundation (Grant No. 465/22).

References

[1] Abtin Afshar, Kai-Min Chung, Yao-Ching Hsieh, Yao-Ting Lin, and Mohammad Mahmoody. 2023. On the (Im)possibility of Time-Lock Puzzles in the Quantum Random Oracle Model. In *ASIACRYPT 2023, Part IV (Lecture Notes in Computer Science, Vol. 14441)*, Jian Guo and Ron Steinfeld (Eds.). Springer, 339–368. doi:10.1007/978-981-99-8730-6_11

[2] James Aspnes, Shlomi Dolev, and Amit Hendin. 2026. Obfuscated Consensus. arXiv:2504.04046 [cs.DC] doi:10.48550/arXiv.2504.04046

[3] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. 2012. On the (im)possibility of obfuscating programs. *J. ACM* 59, 2 (2012), 6:1–6:48. doi:10.1145/2160158.2160159

[4] Michael Ben-Or. 1983. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract). In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, Robert L. Probert, Nancy A. Lynch, and Nicola Santoro (Eds.). ACM, 27–30. doi:10.1145/800221.806707

[5] Shlomi Dolev, Bingyong Guo, Jianyu Niu, and Ziyu Wang. 2024. SodsBC: A Post-Quantum by Design Asynchronous Blockchain Framework. *IEEE Trans. Dependable Secur. Comput.* 21, 1 (2024), 47–62. doi:10.1109/TDSC.2023.3243588

[6] Amos Fiat and Adi Shamir. 1986. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO '86 (Lecture Notes in Computer Science, Vol. 263)*, Andrew M. Odlyzko (Ed.). Springer, 186–194. doi:10.1007/3-540-47721-7_12

[7] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382. doi:10.1145/3149.214121

[8] Rati Gelashvili. 2018. On the optimal space complexity of consensus for anonymous processes. *Distributed Computing* 31, 4 (01 Aug 2018), 317–326. doi:10.1007/s00446-018-0331-9

[9] Aayush Jain, Huijia Lin, and Amit Sahai. 2024. Indistinguishability Obfuscation from Well-Founded Assumptions. *Commun. ACM* 67, 3 (2024), 97–105. doi:10.1145/3611095

[10] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography, Second Edition* (2nd ed.). Chapman & Hall/CRC. doi:10.5555/2700550

[11] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. doi:10.1145/279227.279229

[12] Michael C. Loui and Hosame H. Abu-Amara. 1987. Memory Requirements for Agreement Among Unreliable Asynchronous Processes. In *Parallel and Distributed Computing*, Franco P. Preparata (Ed.). Advances in Computing Research, Vol. 4. JAI Press, 163–183.

[13] Shlomo Moran. 1995. Using approximate agreement to obtain complete disagreement: the output structure of input-free asynchronous computations. In *Third Israel Symposium on the Theory of Computing and Systems*. 251–257. doi:10.1109/ISTCS.1995.377025

[14] National Institute of Standards and Technology. 2015. Secure Hash Standard (SHS). *Federal Information Processing Standards (FIPS) Publication 180-4* 180, 4 (August 2015). https://doi.org/10.6028/NIST.FIPS.180-4

[15] Ronald L. Rivest, Adi Shamir, and David A. Wagner. 1996. *Time-lock puzzles and timed-release crypto.* Technical Report MIT/LCS/TR-684. MIT.