# Blunting an Adversary Against Randomized Concurrent Programs with Linearizable Implementations

Hagit Attiya, Constantin Enea, Jennifer L Welch

# Blunting an Adversary Against Randomized Concurrent Programs with Linearizable Implementations

Hagit Attiya
hagit@cs.technion.ac.il
Technion
Haifa, Israel

Constantin Enea
cenea@irif.fr
LIX, Ecole Polytechnique, CNRS and
Institut Polytechnique de Paris
Palaiseau, France

Jennifer L. Welch
welch@cse.tamu.edu
Texas A&M University
College Station, TX, USA

## ABSTRACT

Atomic shared objects, whose operations take place instantaneously, are a powerful abstraction for designing complex concurrent programs. Since they are not always available, they are typically substituted with software implementations. A prominent condition relating these implementations to their atomic specifications is *linearizability*, which preserves safety properties of the programs using them. However linearizability does not preserve *hyper-properties*, which include probabilistic guarantees of randomized programs: an adversary can greatly amplify the probability of a bad outcome, such as nontermination, by manipulating the order of events inside the implementations of the operations. This unwelcome behavior prevents modular reasoning, which is the key benefit provided by the use of linearizable object implementations. A more restrictive property, *strong linearizability*, does preserve hyper-properties but it is impossible to achieve in many situations.

This paper suggests a novel approach to blunting the adversary's additional power that works even in cases where strong linearizability is not achievable. *We show that a wide class of linearizable implementations, including well-known ones for registers and snapshots, can be modified to approach the probabilistic guarantees of randomized programs when using atomic objects.* The technical approach is to transform the algorithm of each operation of an existing linearizable implementation by repeating a carefully chosen prefix of the operation several times and then randomly choosing which repetition to use subsequently. We prove that the probability of a bad outcome decreases with the number of repetitions, approaching the probability attained when using atomic objects. The class of implementations to which our transformation applies includes the ABD implementation of a shared register using message-passing, the Afek et al. implementation of an atomic snapshot using single-writer registers, the Vitányi and Awerbuch implementation of a multi-writer register using single-writer registers, and the Israeli and Li implementation of a multi-reader register using single-reader registers, all of which are widely used in asynchronous crash-prone systems.


*PODC '22, July 25–29, 2022, Salerno, Italy*

ACM ISBN 978-1-4503-9262-4/22/07…$15.00
https://doi.org/10.1145/3519270.3538446

## CCS CONCEPTS

• **Theory of computation → Distributed computing models**; **Concurrent algorithms**; **Distributed algorithms**; • **Computing methodologies → Distributed algorithms**; **Concurrent algorithms**.

## KEYWORDS

Concurrent Objects; Strong Linearizability; Randomized Programs; ABD Simulation; Atomic Snapshots; Shared registers

**ACM Reference Format:**
Hagit Attiya, Constantin Enea, and Jennifer L. Welch. 2022. Blunting an Adversary Against Randomized Concurrent Programs with Linearizable Implementations. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing (PODC '22), July 25–29, 2022, Salerno, Italy.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3519270.3538446

## 1 INTRODUCTION

Atomic shared objects, whose operations take place instantaneously, are a powerful abstraction for designing complex concurrent programs, as they allow developers to reason about their programs in terms of familiar data structures. Since they are not always available, they are typically substituted with software implementations. A prominent condition relating these implementations to their atomic specifications is *linearizability* [20]. It provides the illusion that processes communicate through shared objects on which operations occur instantaneously in a sequential order, called the *linearization order*, regardless of the actual communication mechanism. A key benefit of linearizability is that it preserves any safety property enjoyed by the program when it is executed with atomic objects.

Unfortunately, linearizability does not preserve *hyper-properties* [11], which include probabilistic guarantees of randomized programs. As demonstrated by examples in [5, 14, 17], an adversary can greatly amplify the probability of a bad outcome, such as nontermination, by manipulating the order of events inside the implementations of the operations. Such behavior invalidates the key benefit of using linearizable objects, which is the modularity that they provide by hiding implementation details behind an interface that mimics atomic behavior. To overcome this drawback, Golab, Higham and Woelfel [14] proposed a more restrictive property, *strong linearizability*, that preserves hyper-properties, including probability distributions. However, not many strongly-linearizable implementations are known and in fact they are impossible in several important cases (cf. Section 6).

This paper suggests a novel approach to blunting the adversary's additional power that works even in cases where strong linearizability is not achievable. To motivate our approach, consider the

well-known ABD [3] linearizable implementation of a read-write register in crash-prone message-passing systems and how it behaves in the context of the simple program given in Algorithm 1, which we distill from the *weakener* program [17]. In the multi-writer version of ABD [22], which we consider throughout and is presented in Algorithm 2, both read and write operations start with a "query" message-exchange phase in which the invoking process obtains the timestamp associated with the most recent value. Then, both operations execute an "update" message-exchange phase; the reader announces the latest value and timestamp before returning the value, while the writer announces the new value and assigns it a larger timestamp. The linearization order of the operations is completely determined by the maximal timestamps that are obtained during the query phases, and thus, their order is determined at the end of the query phase.

**Algorithm 1** Processes $p_0$, $p_1$, and $p_2$ share two registers $R$, written by $p_0$ and $p_1$ and read by $p_2$, and $C$, written by $p_1$ and read by $p_2$.

1: Initially: $R = \perp$, $C = -1$
2: **Code for** $p_i$, $i \in \{0, 1\}$**:**
3: $R := i$
4: if $(i = 1)$ then $C :=$ flip fair coin (0 or 1)
5: **Code for** $p_2$**:**
6: $u_1 := R$; $u_2 := R$; $c := C$
7: if $((u_1 = c) \wedge (u_2 = 1 - c))$ then loop forever
8: else terminate

Algorithm 1 has two processes, $p_0$ and $p_1$, that write their ids to register $R$, then $p_1$ flips a coin and writes the result to another register $C$. A third process, $p_2$, reads $R$ twice and $C$ once; if it succeeds in reading both ids from $R$ and the first id that it reads equals the result of the coin flip, then it loops forever, otherwise it terminates. When the registers are atomic, $p_2$ terminates with probability at least one-half, for any adversary. (See [8] for details.) Yet when the registers are replaced with ABD implementations, a strong adversary [2], which can observe processes' random choices,[1] can interleave the internal steps of the query phase and the steps of the program so as to ensure that $p_2$ *never* terminates. (See [8] for details.)

Instead of attempting to find a strongly-linearizable replacement for ABD, which is impossible [6, 10], we make the key observation that the adversary can disrupt the workings of the program only when the coin flip on Line 4 occurs during the query phase of a read or write operation. The reason is that, after the query phase has completed, the linearization order of the operation is fixed. We also observe that the query phase is "effect-free" in the sense that it can be repeated multiple times without the repetitions interfering with each other or with the behavior of the other processes.

Our modification to ABD is for each operation to *execute the query phase several times, and then randomly choose which one of the values obtained* to use in the rest of the operation. In Algorithm 1, the adversary can make only *one* of these values depend on the result of the coin flip (by scheduling the coin flip during that iteration of the query phase), but that value is used in the rest of the operation with some probability strictly smaller than 1, since values from query phases are chosen uniformly at random. As a result, the program exhibits probabilistic behavior closer to that seen with atomic objects. For example, repeating the query phase twice when ABD is used in Algorithm 1 ensures that $p_2$ terminates with probability at least 1/8, in contrast with the 0 termination probability when using the original ABD implementation. (See [8] for details.) Thus by carefully introducing additional randomization inside the linearizable implementation itself, we blunt the power of the adversary to disrupt the behavior of the randomized program using the object, while keeping the implementation linearizable.

We generalize this idea to develop a transformation for the class of linearizable implementations in which operations can be partitioned, informally speaking, into an *effect-free preamble* followed by a tail. Strong linearizability is only required for executions in which no operation is in the middle of its preamble. The latter property is made precise under the notion of *tail strong linearizability* (Section 3). Our *preamble-iterating transformation* (Section 4.1) repeats the preamble in the implementation of each operation some number of times and then randomly chooses the results of one repetition to use, producing a linearizable implementation of the same object.

*Our main result is that the probability of the program reaching a bad outcome with the transformed objects approaches the probability of reaching the same bad outcome with atomic versions of the objects, as the number of repetitions of the preamble increases relative to the number of random choices made in the program.* Specifically, we show that the probability of the bad outcome using the transformed object is at most the probability of the bad outcome using atomic objects, which is the best case, plus a fraction of the difference between the probabilities of the bad outcome when using the linearizable objects and using the atomic objects. The fraction is the probability that the adversary is able to manipulate the behavior to its advantage, and it decreases as the number of repetitions increases.

Our transformation applies to a broad class of both shared-memory and message-passing implementations that are widely used, and includes ABD (both its original single-writer version [3] and its multi-writer version [22]), the atomic snapshot algorithm [1], the algorithm to construct a multi-writer register using single-writer registers [24], and the algorithm to construct a multi-reader register using single-reader registers [21]. To summarize:

- We introduce a new strengthening of linearizability called *tail strong linearizability* which, roughly speaking, imposes the requirements of strong linearizability only on executions in which each operation has passed its *preamble*. (See Section 3 for the precise definition.) We show that this property is satisfied by a wide range of objects that also have effect-free preambles (Section 5).
- We define a transformation of tail-strongly-linearizable objects with effect-free preambles, which iterates the preamble of each operation multiple times and then randomly chooses an iteration whose results will be used in the rest of the operation (Section 4.1).
- We characterize the blunting power of the "preamble iterated" objects with a quantitative upper bound on the amount by which the probability of reaching a bad outcome increases when using the transformed objects instead of the atomic objects, and relative to using the original linearizable objects (Theorem 4.2 in Section 4.2).

[1] Throughout this paper we consider only strong adversaries and sometimes drop the term "strong".

**Algorithm 2** ABD simulation of a multi-writer register in a message-passing system.

**local variables:**
*sn*, initially 0 {for readers and writers, sequence number used to identify messages}
*val*, initially $v_0$ {for servers, latest register value}
*ts*, initially $(0, 0)$ {for servers, timestamp of current register value, (integer, process id) pair}

**function queryPhase():**
*sn*++
broadcast ⟨"query",*sn*⟩
wait for $\geq \frac{n+1}{2}$ reply msgs to this query msg
(*v*,*u*) := pair in reply msg with largest timestamp
return (*v*,*u*)

**when ⟨"query",*s*⟩ is received from *q*:**
send ⟨"reply",*val*,*ts*,*s*⟩ to *q*

**function updatePhase(*v*,*u*):**
*sn*++
broadcast ⟨"update",*v*, *u*, *sn*⟩
wait for $\geq \frac{n+1}{2}$ ack msgs for this update msg
return

**when ⟨"update",*v*,*u*,*s*⟩ is received from *q*:**
if $u > ts$ then (*val*,*ts*) := (*v*,*u*)
send ⟨"ack",*s*⟩ to *q*

**Read():**
(*v*,*u*) := queryPhase()
updatePhase(*v*,*u*) {write-back}
return *v*

**Write(*v*) for process with id *i*:**
$(-, (t, -))$ := queryPhase() {just need integer in timestamp}
updatePhase($v$,$(t + 1, i)$)
return

## 2 PRELIMINARIES

Randomized programs consist of a number of processes that invoke methods of some set of shared objects, perform local computation, or sample values uniformly at random from a given set of values. We are interested in reasoning about the probability that a strong adversary [2] can cause a program to reach a certain set of program outcomes, defined as sets of values returned by method invocations i.e., operations.

### 2.1 Objects

An *object* is defined by a set of method names and an implementation that defines the behavior of each method. Methods can be invoked in parallel at different processes. In message-passing implementations, processes communicate by sending and receiving messages, while in shared-memory implementations, they communicate by invoking methods of a set of shared objects (e.g., some class of registers) that execute instantaneously (in a single indivisible step), called *base* objects. The pseudo-code we will use to define such implementations can be translated in a straightforward manner to executions seen as sequences of labeled transitions between global states that track the local states of all the participating processes, the states of the shared base objects or the set of messages in transit, depending on the communication model, and the control point of each method invocation in a process. Certain transitions of an execution correspond to initiating a new method invocation, called *call transitions*, or returning from an invocation, called *return transitions*. Such transitions are labeled by call and return actions, respectively. A *call action call* $M(x)_i$ labels a transition corresponding to invoking a method $M$ with argument $x$; $i$ is an identifier of this invocation. A *return action ret* $y_i$ labels a transition corresponding to invocation $i$ returning value $y$. For simplicity, we assume that each method has at most one parameter and at most one return value. We assume that each label of a transition corresponding to a step of an invocation $i$ includes the invocation identifier $i$ and the control point (line number) $\ell$ of that step. In particular, each call transition includes an initial control point $\ell_0$. Such a transition is called a *step* of $i$ at $\ell$.

The set of executions of an object $O$ is denoted by $E(O)$. An execution of an object $O$ satisfies standard well-formedness conditions, e.g., each transition corresponding to returning from an invocation $i$ (labeled by *ret* $y_i$ for some $y$) is preceded by a transition corresponding to invoking $i$ (labeled by *call* $M(x)_i$, for some $M$ and $x$), and for every $i$ there is at most one transition labeled by a call action containing $i$, and at most one transition labeled by a return action containing $i$.

An object where every invocation returns immediately is called *atomic*. Formally, we say that an object $O$ is *atomic* when every transition labeled by *call* $M(x)_i$, for some $M$ and $x$, in an execution (from $E(O)$) is immediately followed by a transition labeled by *ret* $y_i$ for some $y$.

Correctness criteria like linearizability characterize sequences of call and return actions in an execution, called *histories*. The history of an execution $e$, denoted by $hist(e)$, is defined as the projection of $e$ on the call and return actions labeling its transitions. The set of histories of all the executions of an object $O$ is denoted by $H(O)$. Call and return actions *call* $M(x)_i$ and *ret* $y_i$ are called *matching* when they contain the same invocation identifier $i$. A call action is called *unmatched* in a history $h$ when $h$ does not contain the matching return. A history $h$ is called *sequential* if every call *call* $M(x)_i$ is immediately followed by the matching return *ret* $y_i$. Otherwise, it is called *concurrent*. Note that every history of an atomic object is sequential.

### 2.2 (Strong) Linearizability

*Linearizability* [20] defines a relationship between histories of an object and a given set of sequential histories, called a *sequential*

*specification*. The sequential specification can also be interpreted as an atomic object. Therefore, given two histories $h_1$ and $h_2$, we use $h_1 \sqsubseteq h_2$ to denote the fact that there exists a history $h_1'$ obtained from $h_1$ by appending return actions that correspond to some of the unmatched call actions in $h_1$ (completing some pending invocations) and deleting the remaining unmatched call actions in $h_1$ (removing some pending invocations), such that $h_2$ is a permutation of $h_1'$ that preserves the order between return and call actions, i.e., if a given return action occurs before a given call action in $h_1'$ then the same holds in $h_2$. We say that $h_2$ is a *linearization* of $h_1$. A history $h_1$ is called *linearizable* w.r.t. a sequential specification *Seq* iff there exists a sequential history $h_2 \in Seq$ such that $h_1 \sqsubseteq h_2$. An execution $e$ is linearizable w.r.t. *Seq* if $hist(e)$ is linearizable w.r.t. *Seq*. An object $O$ is linearizable w.r.t. *Seq* iff each history $h_1 \in H(O)$ is linearizable w.r.t. *Seq*.

Two objects $O_1$ and $O_2$ are called *equivalent* when they are linearizable w.r.t. the same sequential specification *Seq* and for every history $h \in Seq$, $H(O_1)$ contains a history linearizable w.r.t. $h$ iff $H(O_2)$ contains a history linearizable w.r.t. $h$.

*Strong linearizability* [14] is a strengthening of linearizability that enables preservation of probability distributions in randomized programs using a certain object $O$ instead of an *atomic* object equivalent to $O$. It also enables preservation of more generic hypersafety properties [5]. A set of executions $E \subseteq E(O)$ of an object $O$ is called *strongly linearizable* when it admits linearizations that are consistent with linearizations of prefixes that belong to $E$ as well. Formally, $E$ is strongly linearizable w.r.t. a sequential specification *Seq* iff there exists a function $f : E \rightarrow Seq$ such that:

- for any execution $e \in E$, $hist(e) \sqsubseteq f(e)$, and
- $f$ is prefix-preserving, i.e., for any two executions $e_1, e_2 \in E$ such that $e_1$ is a prefix of $e_2$, $f(e_1)$ is a prefix of $f(e_2)$.

An object is called *strongly linearizable* when its entire set of executions $E(O)$ is strongly linearizable.

## 2.3 Randomized Programs

A *program* $P(\mathcal{O})$ is composed of a number of processes that invoke methods on a set of shared objects $\mathcal{O}$. Besides shared object invocations, a process can also perform some local computation (on some set of local variables), and use an instruction random$(V)$, where $V$ is a subset of a domain of values $\mathbb{V}$, to sample a value from $V$ uniformly at random. This value can be used, for instance, as an input to a method invocation. The syntax used for local computation instructions is not important, and we omit a precise formalization.

An *execution* of a program $P(\mathcal{O})$ is an interleaving of steps taken by the processes it contains. A step can correspond to either

- an interaction with a shared object in $\mathcal{O}$, i.e., a method invocation, internal step of an object implementation, or returning from a method, or
- a local computation in the program, e.g., an execution of random$(V)$, for some $V$.

As expected, the sequence of steps in an execution follows the control-flow in each process and the internal behavior of the shared objects in $\mathcal{O}$ (whether they be implemented on top of a message-passing or shared-memory system).

The *outcome* of a program execution is a mapping from shared object method invocations to the values they return in that execution. In order to relate outcomes in different executions of the same program $P(\mathcal{O})$, we assume that shared object method invocations in executions of $P(\mathcal{O})$ have unique identifiers that relate to the *syntax* of $P(\mathcal{O})$. These identifiers can be defined, for instance, as a triple of a process id, the control point (line number) at which that invocation occurs, and the number of times this control point occurred in the past (in order to deal with looping constructs). Then, an outcome maps these identifiers to return values. An outcome of a program $P(\mathcal{O})$ is the outcome of an execution of $P(\mathcal{O})$.

Consider two sets of objects $\mathcal{O}_1$ and $\mathcal{O}_2$ for which there exists a bijection $\lambda$ that maps each object $O \in \mathcal{O}_1$ to an *equivalent* object $O' \in \mathcal{O}_2$. Given a program $P(\mathcal{O}_1)$, the program $P(\mathcal{O}_2)$ is obtained by substituting every object $O \in \mathcal{O}_1$ with the corresponding object $\lambda(O) \in \mathcal{O}_2$.

Proposition 2.1. *$P(\mathcal{O}_1)$ and $P(\mathcal{O}_2)$ have the same set of outcomes.*

## 2.4 Adversaries

We say that a program execution *observes* a sequence of random values $\vec{v}$ if the $i$-th occurrence of a step that samples a random value (by executing a random$(V)$ instruction) returns $\vec{v}[i]$, where $\vec{a}[i]$ is the $i$-th position in a vector $\vec{a}$. A *schedule* is a sequence of process ids. An execution *follows* a schedule $\vec{s}$ when the $i$-th step of the execution is executed by the process $\vec{s}[i]$. In the following, we assume *complete* schedules that make the program terminate. We denote by $e[P(\mathcal{O}), \vec{v}, \vec{s}]$ the unique execution of a program $P(\mathcal{O})$ that observes $\vec{v}$ and follows $\vec{s}$.

For a program $P(\mathcal{O})$, a *(strong) adversary* $A$ against $P(\mathcal{O})$ is a mapping from sequences of values in $\mathbb{V}$ to complete schedules. We assume that for every two sequences $\vec{v_1}, \vec{v_2} \in \mathbb{V}^*$ that have a common prefix of length $m$, the executions $e[P(\mathcal{O}), \vec{v_1}, A(\vec{v_1})]$ and $e[P(\mathcal{O}), \vec{v_2}, A(\vec{v_2})]$ are the same until the $(m+1)$-th occurrence of a step that samples a random value, or the end of the execution if no such steps remain. This assumption captures the constraint that the scheduling decisions of a strong adversary do not depend on future randomized choices. A strong adversary $A$ defines a set of executions $E(A)$, each of which observes a sequence of values $\vec{v}$ and follows the schedule $A(\vec{v})$.

An adversary $A$ against $P(\mathcal{O})$ defines a probability distribution over program outcomes (of executions in $E(A)$), denoted by $OutDist(P(\mathcal{O}), A)$. Given a set of outcomes $\mathcal{B}$, $Prob[P(\mathcal{O})||A \rightarrow \mathcal{B}]$ is the probability defined by $OutDist(P(\mathcal{O}), A)$ of an outcome being contained in $\mathcal{B}$. The *probability of* $P(\mathcal{O})$ *reaching* $\mathcal{B}$, denoted by $Prob[P(\mathcal{O}) \rightarrow \mathcal{B}]$, is defined as the maximal probability $Prob[P(\mathcal{O})||A \rightarrow \mathcal{B}]$ over all possible adversaries $A$. In the context of our results, the set of outcomes $\mathcal{B}$ is interpreted as some set of "bad" states, and the goal is to minimize the probability of a program reaching them.

The following result shows that a program using atomic objects minimizes the probability of reaching a set of outcomes, among programs where the atomic objects can be replaced with equivalent ones. This follows from the fact that an adversary can restrict itself to schedules where each method invocation is executed in isolation (a method can be called only when there is no other pending call),

and the outcomes obtained in executions following such schedules can also be obtained with executions of atomic objects. For a set of objects $\mathcal{O}$, $\mathcal{O}_a$ is the set of *atomic* objects $O'$ that are equivalent to objects $O \in \mathcal{O}$.

Proposition 2.2. *For any program $P(\mathcal{O})$ and set of outcomes $\mathcal{B}$, $Prob[P(\mathcal{O}) \rightarrow \mathcal{B}] \geq Prob[P(\mathcal{O}_a) \rightarrow \mathcal{B}]$.*

Algorithm 1 is an example of a program $P$ where $Prob[P(\mathcal{O}) \rightarrow \mathcal{B}]$ is strictly greater than $Prob[P(\mathcal{O}_a) \rightarrow \mathcal{B}]$, as discussed in the introduction. In this case, $\mathcal{O}$ consists of two instances of the ABD register, one for $R$ and one for $C$, and $\mathcal{B}$ is the set of outcomes where the return values of $p_2$'s invocations satisfy $u_1 = c$ and $u_2 = 1 - c$. These values make $p_2$ not terminate.

The two probabilities in Proposition 2.2 are equal when $\mathcal{O}$ is a set of strongly linearizable objects:

Theorem 2.3 ([14]). *For any program $P(\mathcal{O})$ using a set of strongly linearizable objects $\mathcal{O}$, and set of outcomes $\mathcal{B}$, $Prob[P(\mathcal{O}) \rightarrow \mathcal{B}] = Prob[P(\mathcal{O}_a) \rightarrow \mathcal{B}]$.*

## 3 TAIL STRONG LINEARIZABILITY

We define a generalization of strong linearizability, called *tail strong linearizability*, which requires that executions be mapped to prefix-preserving linearizations only when each method invocation has executed a minimal number of steps called a *preamble*. The relationship between linearizations of different executions where some invocation has *not* executed its preamble fully is unconstrained. When the preamble of every invocation is "empty" (i.e., it includes only the call transition), this becomes the standard notion of strong linearizability. When the preamble of every invocation is "full" (i.e., it includes all the steps of the invocation), this is equivalent to standard linearizability (since linearizability requires anyway that any invocation $i$ is linearized before any other invocation $i'$ that starts after $i$ returns). Section 4 defines a preamble-iterating transformation of tail strongly linearizable objects that limits the increase in the probability of a bad outcome when a program uses the transformed objects instead of equivalent atomic objects.

Let $O$ be an object with a set of methods $\mathbb{M}$. A *preamble mapping* $\Pi$ of $O$ is a mapping that associates each method $M \in \mathbb{M}$ with a control point $\ell$ representing the last step of its preamble. We assume that every control-flow path of $M$ should pass through $\ell$ and that $\ell$ can be reached only once (it is not inside the body of a loop). The trivial preamble mapping that associates each method to the initial control point $\ell_0$ is denoted by $\Pi_0$. For instance, for the multi-writer version of ABD (listed in Algorithm 2 and described in the introduction), we are interested in a preamble mapping that associates the **Read** and **Write** methods with the control points where the value with the largest timestamp received from responses to query messages is assigned (Lines 22 and 26, respectively, in Algorithm 2).

Given an execution $e$ and a method invocation $i$, we say that $i$ *passed* a control point $\ell$ when $e$ contains a step of $i$ at $\ell$. An execution $e$ is *complete* w.r.t. a preamble mapping $\Pi$ if each invocation of a method $M$ in $e$ passed the control point $\Pi(M)$. The set of executions of $O$ complete w.r.t. $\Pi$ is denoted by $E(O, \Pi)$.

An object $O$ is called *tail strongly linearizable* w.r.t. a preamble mapping $\Pi$ and a sequential specification *Seq* when it is linearizable w.r.t. *Seq* and the set of executions $E(O, \Pi)$ is strongly linearizable w.r.t. *Seq*. Note that strong linearizability is equivalent to tail strong linearizability w.r.t. $\Pi_0$.

When reasoning about programs that use more than one object, we rely on the fact that tail strong linearizability is *local* in the sense that it holds for the union of a set of objects that are each tail strongly linearizable. Locality holds for tail strong linearizability as a straightforward consequence of the fact that standard strong linearizability is local [14].

Theorem 3.1. *A set of histories $H$ of executions with multiple objects $O_1,\ldots,O_m$ is tail strongly linearizable w.r.t. some preamble mapping $\Pi_1 \cup \ldots \cup \Pi_m$, where $\Pi_j$ is a preamble mapping of $O_j$, iff for all $j$, $1 \leq j \leq m$, the set $H_j = \{h|O_j : h \in H\}$, where $h|O_j$ is the projection of $h$ on call and return actions of $O_j$, is tail strongly linearizable w.r.t. $\Pi_j$.*

## 4 BLUNTING AN ADVERSARY AGAINST TAIL STRONGLY LINEARIZABLE OBJECTS

We define a methodology for transforming tail strongly linearizable objects whose preambles have a certain property we call "effect-free" into equivalent objects. The use of the transformed objects can reduce the probability that a program using the objects reaches a set of (bad) outcomes. Intuitively, the transformed objects can blunt the power of any adversary against a program using them and in the limit restrict its power to what it has when the program uses atomic objects (which is a lower bound by Proposition 2.2). As we show in Section 5, the class of objects to which the transformation applies includes a broad set of widely-used objects, including the ABD register (both its original single-writer version [3] as well as the multi-writer version [22]), the atomic snapshot algorithm using single-writer registers of Afek et al. [1], the Vitányi and Awerbuch algorithm to construct a multi-writer register from single-writer registers [24], and the Israeli and Li algorithm to construct a multi-reader register from single-reader registers [21]. None of these implementations is strongly linearizable and in fact strongly-linearizable implementations are known to be impossible in most of these cases (see Section 6).

### 4.1 The Preamble-Iterating Transformation for Tail Strongly Linearizable Objects

The preamble-iterating transformation is defined in Algorithm 3. For a given integer $k \geq 1$, object $O$, and preamble mapping $\Pi$, we define an object $O^k_\Pi$ (we may omit the preamble mapping $\Pi$ from the notation when it is understood from the context) where each method $M$ is replaced with a method $M^k$ that iterates the preamble of $M$ (see the **for** loop in Algorithm 3) $k$ times and uses the values of a randomly chosen iteration for the rest of the code (corresponding to index $j$ selected in Line 8). To simplify the notations, we assume that the code of each preamble of a method $M$ (the code up to and including the control point $\Pi(M)$) is encapsulated in a function called PREAMBLE that takes the same input as $M$ and returns the values of $M$'s local variables after executing that preamble. These values are stored in the array *locals*. The rest of the code, which uses the values in *locals*, is left unchanged. The results of the preamble

iterations are stored in a two dimensional array $\overrightarrow{locals}$ where each row has the same size as *locals*.

This transformation leads to an equivalent object provided that the preamble contains only *effect-free* computation, which, informally speaking, does not affect the behavior of the other processes running concurrently (effect-free computation can affect the state of the process that executes it). For instance, the preamble of ABD's **Read** and **Write** methods consists in sending "query" messages to the other processes, waiting for replies, and computing the largest timestamp value from the replies (the queryPhase function in Algorithm 2). Sending a reply to a query message from another concurrently running process does not affect the behavior of the sender, as its local variables remain unchanged (cf. Algorithm 2).

**Algorithm 3** Transforming a tail strongly linearizable object $O$ to $O^k$, $k \geq 1$. Each method $M$ of $O$ is transformed to a method $M^k$ of $O^k$.

```
1:  method M(v):
2:  locals := PREAMBLE(v)
3:  // rest of the code ...

4:  method M^k(v):
5:  for i := 1 to k do
6:      locals→[i] := PREAMBLE(v)
7:  end for
8:  j := random([1..k])
9:  locals := locals→[j]
10: // rest of the code ...
```

In general, a computation step of an object implementation is either

- an invocation to a method of a base object, e.g., a register, which is assumed to be *atomic*, or
- a *send/receive* step in the context of a message-passing system, or
- a local computation step on some set of local variables (which cannot be accessed by other processes).

A computation step is called *effect-free* if it is a local computation step, or, if in the first case, the invoked method itself is effect-free, e.g., a **Read** method of an atomic register, or if in the second case, it is a receive or a send of a message that does not modify the local state of the receiving process, e.g., sending a "query" message in the ABD register. For a preamble mapping $\Pi$, we say that a method $M$ has an *effect-free preamble* if all the computation steps up to and including $\Pi(M)$ are effect-free. An object is said to have *effect-free preambles* iff all its methods have effect-free preambles.

It can be easily proved that $O^k$ is equivalent to $O$, provided that $O$ has effect-free preambles. We also assume that *the original tail strongly linearizable objects are deterministic*, i.e., they do not rely on randomization. Indeed, by definition, repeating the effect-free preamble has no effect on local states of other processes. Each execution of $O^k$ can be transformed to an execution of $O$ where all the preamble repetitions that are not "used" in an invocation (i.e., the value they compute is not selected to continue the computation) can be simply removed. Since the original $O^k$ execution has exactly the same history as the one of $O$, its linearizability w.r.t. the specification of $O$ follows from the linearizability of the execution of $O$. Conversely, every execution of $O$ can be transformed to an execution of $O^k$ by "appending" sufficiently many repetitions of the preamble and restricting the random choice to select the first repetition.

Theorem 4.1. *For every object $O$ with effect-free preambles and $k \geq 1$, $O^k$ is equivalent to $O$.*

## 4.2 Quantifying the Blunting Power

We characterize the power of $O^k$ objects in lowering the probability that a program $P$ using them reaches some set $\mathcal{B}$ of outcomes, compared to $P$ using the original objects $O$ instead. Since we interpret $\mathcal{B}$ as "bad" outcomes, lowering this probability is desirable.

For a set of objects $\boldsymbol{O}$, $\boldsymbol{O}^k$ is the set of objects $O^k$ with $O \in \boldsymbol{O}$. While stating the result below, the program $P$ and the set of outcomes $\mathcal{B}$ are fixed (but arbitrary), and to simplify the notation, we write $Prob[\boldsymbol{O}]$ instead of $Prob[P(\boldsymbol{O}) \to \mathcal{B}]$, for any set of objects $\boldsymbol{O}$. Also, we say that a program $P(\boldsymbol{O})$ has *at most $r$* random *steps* if every execution of $P$ contains at most $r$ steps corresponding to executing a random instruction. This definition applies to programs using objects $\boldsymbol{O}$ and not the transformed objects $\boldsymbol{O}^k$ which introduce additional random steps.

We show that $Prob[\boldsymbol{O}^k]$ decreases with respect to $Prob[\boldsymbol{O}]$ as the number of preamble iterations $k$ increases and exceeds the maximum number $r$ of random steps in the program. This provides a trade-off between time complexity, which grows with $k$, and the probability of reaching bad outcomes, which decreases with $k$. This result is based on a worst-case analysis which makes no assumptions about the structure of the program.

Theorem 4.2. *For every program $P(\boldsymbol{O})$ with $n \geq 1$ processes and at most $r \geq 1$* random *steps, where $\boldsymbol{O}$ is a set of deterministic tail strongly linearizable objects with effect-free preambles, for every set of outcomes $\mathcal{B}$, and for every positive integer $k$,*

$$Prob[\boldsymbol{O}^k] \leq Prob[\boldsymbol{O}_a] + \left[1 - \left(\frac{\max\{0, k-r\}}{k}\right)^{n-1}\right] \cdot (Prob[\boldsymbol{O}] - Prob[\boldsymbol{O}_a]) .$$

Theorem 4.2 states that the probability of a bad outcome when using objects in which the preamble is iterated $k$ times is at most the probability when using atomic objects plus a fraction of the difference between the probabilities when using atomic objects and when using the original linearizable objects. The fraction is, roughly speaking, the probability that the adversary is able to manipulate the behavior to its advantage, and it goes to 0 as $k$ increases. Thus the probability with the preamble-iterated objects approaches the probability with atomic objects.

## 4.3 Proof Outline for Theorem 4.2

We start by introducing some terminology. The program $P(\boldsymbol{O}^k)$ has two types of random instructions: the random instructions coming from the original program $P(\boldsymbol{O})$, which are outside of object implementations, and the random instructions added in the $\boldsymbol{O}^k$ implementations (see Algorithm 3). The former are called *program* random instructions, and the latter *object* random instructions. Steps in an execution corresponding to program (object) random instructions are called program (object) random steps. Each method

invocation in an execution of $P(O^k)$ performs $k$ iterations of a preamble (of some method of an object in $O$). A preamble iteration is called *randomization-free* when it does *not* overlap with a program random step, i.e., every program random step occurs either before or after all the steps of that preamble iteration.

Let $A$ be an adversary against $P(O^k)$ defining a probability distribution over executions/outcomes. Let $X$ be the event that *all* the object random steps return indices that correspond to randomization-free preamble iterations. We decompose the probability of $A$ reaching a set of outcomes $\mathcal{B}$ by conditioning on $X$:

$$Prob[P(O^k)||A \to \mathcal{B}] = Prob[(P(O^k)||A \to \mathcal{B}) \mid X] \cdot Prob[X] + Prob[(P(O^k)||A \to \mathcal{B}) \mid \neg X] \cdot (1 - Prob[X]) \quad (1)$$

Lemma 4.3 (proved below) shows that the probability of $A$ reaching $\mathcal{B}$ conditioned on $X$ is upper bounded by the probability of any adversary reaching $\mathcal{B}$ in the same program but with atomic objects instead of $O^k$. That is, $Prob[(P(O^k)||A \to \mathcal{B}) \mid X] \leq Prob[P(O_a) \to \mathcal{B}]$. Lemma 4.4 (proved below) shows that the probability of reaching $\mathcal{B}$ with $O^k$ conditioned on $\neg X$ cannot be larger than the probability of reaching $\mathcal{B}$ with $O$, i.e., $Prob[(P(O^k)||A \to \mathcal{B}) \mid \neg X] \leq Prob[P(O) \to \mathcal{B}]$. Substituting into (1), we get that

$$\begin{aligned} Prob[P(O^k)||A \to \mathcal{B}] \leq & Prob[P(O_a) \to \mathcal{B}] \cdot Prob[X] \\ & + Prob[P(O) \to \mathcal{B}] \cdot (1 - Prob[X]) \quad (2) \\ = & Prob[P(O_a) \to \mathcal{B}] \\ & + (1 - Prob[X]) \cdot (Prob[P(O) \to \mathcal{B}] - Prob[P(O_a) \to \mathcal{B}]) \end{aligned}$$

Lemma 4.5 (proved below) shows that $Prob[X] \geq \left(\frac{\max\{0, k-r\}}{k}\right)^{n-1}$, which concludes the proof of the theorem.

## 4.4 Detailed Proofs

Lemma 4.3. $Prob[(P(O^k)||A \to \mathcal{B}) \mid X] \leq Prob[P(O_a) \to \mathcal{B}]$.

Proof. Based on the adversary $A$, we will define an adversary $A_O$ against $P(O)$ that mimics the adversary $A$ against $P(O^k)$ conditioned on $X$ for program random steps and takes the choices for object random steps that maximize the probability of reaching $\mathcal{B}$. $A_O$ will cause all the prefixes of executions in $E(A_O)$ that end with a program random step to be complete w.r.t. each preamble mapping of an object in $O$. The construction of $A_O$ will ensure that

$$Prob[(P(O^k)||A \to \mathcal{B}) \mid X] \leq Prob[P(O)||A_O \to \mathcal{B}] \quad (3)$$

Then, we will use the completeness w.r.t. preamble mappings of execution prefixes to show that

$$Prob[P(O)||A_O \to \mathcal{B}] \leq Prob[P(O_a) \to \mathcal{B}]. \quad (4)$$

which will complete the proof. Details follow.

Given a sequence $\vec{v}$ of values returned by program random steps, let $\vec{u}$ be a sequence of values returned by program or object random steps such that $\vec{v}$ is a subsequence of $\vec{u}$ and for every index $i$ in $\vec{u}$ representing the value of an object random step,

$$\begin{aligned} & Prob[(P(O^k)||A \to \mathcal{B}) \mid X \mid \vec{u}[\leq i]] \\ & = max_{v \in \mathbb{V}} Prob[(P(O^k)||A \to \mathcal{B}) \mid X \mid \vec{u}[\leq i-1] \cdot v] \quad (5) \end{aligned}$$

where $Prob[(P(O^k)||A \to \mathcal{B}) \mid X \mid \sigma]$ is the probability that $A$ reaches $\mathcal{B}$ in $P(O^k)$ conditioned on $X$, and further conditioned on the fact that the first $|\sigma|$ (program or object) random steps return the values in $\sigma$ (in the order defined by $\sigma$), and $\vec{u}[\leq i]$ is the prefix of $\vec{u}$ of length $i$ (by convention, $\vec{u}[\leq -1]$ is the empty sequence $\epsilon$). The schedule $A(\vec{u})$ contains $k$ preamble iterations for each method invocation, but only one of them, determined by the result of the object random step in that invocation, is used to continue the computation. Let $\mathsf{remRedundant}(A(\vec{u}))$ be the schedule where all the $k-1$ preamble iterations that are not used in a method invocation are removed. By the definition of the $O^k$ objects, $\mathsf{remRedundant}(A(\vec{u}))$ is a schedule producing a valid execution of $P(O)$. We define

$$A_O(\vec{v}) = \mathsf{remRedundant}(A(\vec{u})).$$

By the construction, property (5) in particular, we have that property (3) holds. Also, since we consider schedules of $A$ conditioned on $X$, all the preamble iterations selected by object random steps are randomization-free, and therefore, at every program random step in $\mathsf{remRedundant}(A(\vec{u}))$, there is no invocation that started but did not finish its preamble.

To prove property (4), we show that there exists an adversary $A_{O_a}$ against $P(O_a)$ such that $OutDist(P(O), A_O) = OutDist(P(O_a), A_{O_a})$. We rely on the facts that each object in $O$ is tail strongly linearizable, that tail strong linearizability is local (cf. Theorem 3.1), and that all the prefixes of executions in $E(A_O)$ ending with a program random step are complete w.r.t. each preamble mapping of an object in $O$. The adversary $A_{O_a}$ is defined iteratively by enumerating program random steps. Initially, by the definition of an adversary, all executions produced by $A_O$ are identical until the first occurrence $rs_1$ of a program random step. By tail strong linearizability, it is possible to define a valid linearization (satisfying each object specification) of the invocations that start before $rs_1$ which does not depend on execution steps that follow $rs_1$ (i.e., this linearization can be extended by appending more invocations when considering steps after $rs_1$). Let $\sigma_0$ be such a linearization. We impose the constraint that all the executions produced by $A_{O_a}$ start with $\sigma_0$.

Next, we focus on execution prefixes that end just before the second occurrence $rs_2$ of a program random step. Assume that $rs_1$ is a random choice between a set of values $V$ and let $v \in V$. Using again the definition of an adversary, all the executions produced by the restriction of $A_O$ to the domain $v \cdot \mathbb{V}^*$ (sequences of values starting with $v$) are identical until $rs_2$. By tail strong linearizability, there exists a linearization $\sigma_v$ of the invocations that started before $rs_2$ in these executions such that $\sigma_0$ is a prefix of $\sigma_v$. Moreover, $\sigma_v$ can be chosen in such a way that it does not depend on execution steps that follow $rs_2$. We define $A_{O_a}$ such that $A_{O_a}(v \cdot \mathbb{V}^*) \in \sigma_v \cdot Act^*$ for each $v \in V$ ($Act$ denotes the set of call/return actions in a history). That is, each execution that the adversary produces when the first program random step returns $v$ starts with the linearization $\sigma_v$.

Iterating the same construction for all the remaining program random steps, we get an adversary $A_{O_a}$ against $P(O_a)$ such that $A_{O_a}(\vec{v})$ is a linearization of the invocations in $A_O(\vec{v})$, for all $\vec{v}$. Therefore, $OutDist(P(O), A_O) = OutDist(P(O_a), A_{O_a})$, and property (4) holds. □

Lemma 4.4. $Prob[(P(O^k)||A \to \mathcal{B}) \mid \neg X] \leq Prob[P(O) \to \mathcal{B}]$.

Proof. As in the proof of Lemma 4.3, property (3), one can define an adversary $A'_O$ against $P(O)$ that mimics the adversary $A$ against $P(O^k)$ for program random steps and takes the choices for object random steps that maximize the probability of reaching $\mathcal{B}$. This argument is actually agnostic to the conditioning on $\neg X$, because it does not depend on the specific results returned by object random steps from which to make the desired choice. We include the conditioning only to match the proof goal coming from (1). We have that

$$Prob[(P(O^k)||A \rightarrow \mathcal{B}) \mid \neg X] \leq Prob[P(O)||A'_O \rightarrow \mathcal{B}] \quad (6)$$

The result follows from the fact that $Prob[P(O)||A'_O \rightarrow \mathcal{B}] \leq Prob[P(O) \rightarrow \mathcal{B}]$. □

Lemma 4.5. $Prob[X] \geq \left(\frac{\max\{0,k-r\}}{k}\right)^{n-1}$.

Proof. Since the random choices in $O^k$ method invocations are independent, we have that $Prob[X] = \prod_i Prob[X_i]$ where $X_i$ is the event that the $i$-th object random step in an invocation to a method of $O^k$ chooses a randomization-free preamble iteration (we assume an arbitrary but fixed total order on invocations in $P$). The minimal value for $Prob[X]$ can be attained by making many $Prob[X_i]$ as small as possible. To minimize the product of $Prob[X_i]$ terms, we need that each program random step overlaps with a maximum number of preamble iterations, i.e., one preamble iteration from each other process. Then, to maximize the number of small $Prob[X_i]$ terms, we need to maximize the number of invocations that contain a maximal number of preamble iterations overlapping with a program random step. These two constraints can be attained assuming that all program random steps are in the same process and each one of them overlaps with a different preamble iteration from the same invocation of each other process. If $k \leq r$, in the worst case the adversary might be able to ensure that no object random step returns an index that corresponds to a randomization-free preamble iteration, which is the reason for the use of the max function. Therefore, for $n-1$ invocations $i$,

$$Prob[X_i] = \frac{\max\{0, k-r\}}{k}$$

and $Prob[X_j] = 1$ for the rest of the invocations $j$. Therefore,

$$Prob[X] \geq \left(\frac{\max\{0, k-r\}}{k}\right)^{n-1}$$

□

# 5 EXAMPLES OF TAIL STRONGLY LINEARIZABLE OBJECTS

We discuss several objects introduced in the literature that are *not* strongly linearizable, but are tail strongly linearizable with respect to some non-trivial, effect-free preamble mapping.

## 5.1 ABD Register

Variations of the ABD implementation of a register in a crash-prone message-passing system are used in many applications. Unfortunately, it is impossible to have a strongly linearizable version of ABD [6, 10]. However, as we show next, our transformation is applicable to ABD.

Specifically, we show that the multi-writer variant [22] of the ABD register [3] (which is listed in Algorithm 2 and explained in the introduction) is tail strongly linearizable w.r.t. the preamble mapping $\Pi_{ABD}$ that associates **Read** and **Write** with the control points Lines 22 and 26, respectively. These are the control points of the steps that assign the return value of queryPhase to $(v, u)$ and $(-, (t, -))$, respectively.

Theorem 5.1. *The ABD object in Algorithm 2 is tail strongly linearizable w.r.t.* $\Pi_{ABD}$.

Proof. The timestamp of a **Read** invocation is the timestamp returned by its query phase (the value $u$ at line 22), and the timestamp of a **Write** is the timestamp given as parameter to its update phase (the pair $(t+1, i)$ at line 27). The timestamp of an invocation $o$ is denoted by $\text{ts}(o)$.

Given an execution $e$ that is complete w.r.t. $\Pi_{ABD}$, we say that an invocation $o$ is *logically-completed* in $e$ when there exists an invocation $o'$ that returns in $e$ such that $\text{ts}(o) \leq \text{ts}(o')$. Since $o$ and $o'$ may coincide, if an invocation returns in $e$, then it is also logically-completed in $e$. By definition, every invocation in $e$ has a well-defined timestamp (since every invocation passed the query phase).

We define a function $f$ that associates to each such execution $e$ a linearization that contains all the invocations that are logically-completed in $e$ ordered according to their timestamp. A set of invocations in $e$ that have the same timestamp consists of exactly one **Write** invocation and some number of **Read** invocations. The linearization $f(e)$ orders the write before all the reads with the same timestamp, if any.

To show that $f$ is prefix-preserving, let $e, e' \in E(\text{ABD}, \Pi)$ such that $e$ is a prefix of $e'$. We show that a linearization of $e$ where invocations that are logically-completed in $e$ are ordered before invocations that are *not* logically-completed is consistent with an analogous linearization of $e'$.

For an invocation $o_1$ that is logically-completed in $e$, we show that $\text{ts}(o_1) \leq \text{ts}(o_2)$ for every invocation $o_2$ that is not logically-completed in $e$. There are two cases to consider. First, if $o_2$ queries after $e$, then we use the fact that ABD guarantees that the timestamp of an invocation is smaller than or equal to the timestamp returned by any query phase starting after that invocation returned. By the definition of logically-completed, there exists an invocation $o'_1$ that returns in $e$ such that $\text{ts}(o_1) \leq \text{ts}(o'_1)$. Using the property of ABD mentioned above, we get that $\text{ts}(o'_1) \leq \text{ts}(o_2)$, which implies that $\text{ts}(o_1) \leq \text{ts}(o_2)$. Second, if $o_2$ queries during $e$, then by the definition of logically-completed, $\text{ts}(o_2) > \text{ts}(o'_2)$ for every invocation $o'_2$ that returns in $e$. Since $o_1$ is logically-completed in $e$, we get that there exists an invocation $o'_1$ that returns in $e$ such that $\text{ts}(o_1) \leq \text{ts}(o'_1)$. Therefore, $\text{ts}(o_1) \leq \text{ts}(o_2)$. Next, we show that there cannot exist a **Write** invocation $o_1$ that is *not* logically-completed in $e$ while a **Read** invocation $o_2$ with the same timestamp is logically-completed in $e$. Clearly, $o_1$ cannot query after $e$ since $o_2$ queries during $e$ by definition. Assuming that both invocations query during $e$, we get a contradiction because the definition of logically-completed implies that $\text{ts}(o_1) > \text{ts}(o'_1)$ for every invocation $o'_1$ that returns in $e$ and there exists an invocation $o'_2$ that returns in $e$ such that $\text{ts}(o_2) \leq \text{ts}(o'_2)$. These two statements imply that $\text{ts}(o_1) > \text{ts}(o_2)$

which is a contradiction to the fact that $o_1$ and $o_2$ have the same timestamp.

Finally, an invocation $o_1$ that is *not* logically-completed in $e$ cannot return before an invocation $o_2$ that is logically-completed in $e$. Since $o_2$ queries during $e$, this would imply that $o_1$ returns in $e$ which would imply that $o_1$ is logically-completed in $e$. □

The above result holds also for the original single-writer version [3], which is also not strongly linearizable [10, 16].

## 5.2 Snapshot

Another popular shared object is the atomic snapshot. It is impossible to implement a strongly-linearizable lock-free snapshot object using single-writer registers [19] and it is impossible to implement a strongly-linearizable wait-free snapshot object using multi-writer registers [12]. However, we show next that we can apply our transformation to the linearizable wait-free snapshot implementation in [1], which uses single-writer registers.

The snapshot object implementation of [1] uses an array M of registers whose length is the number of processes (accesses to these registers are atomic, i.e., they execute instantaneously). It provides a **Scan** method that returns a snapshot of the array and an **Update**($v$) method by which a process $i$ writes value $v$ in M[$i$]. **Scan** performs a series of *collects*, i.e., successive reads of the array's cells in some fixed order; a collect in a process can interleave with steps of other processes. This series of collects stops when either two successive collects return identical values, or the process observes that another process has executed at least two **Update** invocations during the timespan of the **Scan**. In the latter case, the return value is the last snapshot written by the other process during an **Update**. An **Update** invocation at a process $i$ starts with a **Scan** followed by an atomic write to M[$i$] of the result of **Scan** together with the value received as argument (and a local sequence number $\text{seq}_i$ that is read in other **Scan** invocations).

This snapshot object is known to *not* be strongly linearizable [14], but it is tail strongly linearizable w.r.t. a preamble mapping that maps each **Scan** to the control point just before it returns and each **Update** to the initial control point. The linearization associated to an execution that is complete w.r.t. this preamble mapping contains all the (possibly pending) **Scan** invocations and all the **Update** invocations that performed their writes to the array cells, in some order consistent with the specification (each **Scan** is linearized after an **Update** if it observes its value). Actually, the preamble of **Update** can be defined in an arbitrary manner, e.g., extended until the end of its scan, and tail strong linearizability would still hold. The reason is that an **Update** is linearized only if it executed its write— the scan it performs before the write is only to ensure progress (wait-freedom). As can be seen in Section 4, extending a preamble may help in reducing the probability of reaching "bad" outcomes, but this comes at a cost in terms of time complexity.

## 5.3 Multi-Writer Multi-Reader Register

Another central shared object is a multi-writer multi-reader register. There is no strongly-linearizable wait-free implementation of such a register using single-writer registers [19]. We show, however, that our transformation can be applied to the linearizable implementation in [24].

In this implementation, each value written has a timestamp, which is a pair consisting of an integer and a process identifier. A single-writer register *Val*[$i$] is associated with each writer $i$ of the implemented register. When a **Read** is invoked on the implemented register, the reader reads (value, timestamp) pairs from all the *Val* registers, chooses the value with the largest timestamp using lexicographic ordering, and returns that value. When a **Write** of value $v$ on the implemented register is invoked at writer $i$, the writer calculates a new timestamp and writes the value together with the timestamp into *Val*[$i$]. To calculate the new timestamp, $i$ reads all the *Val* variables and extracts from it the timestamp entry. Its new timestamp is one plus the maximal timestamp of all other processes, together with $i$'s identifier. This implementation is tail strongly linearizable by choosing the preamble of the **Read** method to end just before it returns and the preamble of the **Write** method to end immediately before writing to *Val*[$i$]. The tail strong linearizability proof is similar to the one for the ABD register.

## 5.4 Single-Writer Multi-Reader Register

Yet another standard shared object is a (single-writer) multi-reader register. A well-known implementation of such a register using (single-writer) single-reader registers is given in [21]. This implementation is not strongly linearizable, which can be shown by mimicking the counter-example for the ABD register appearing in [16]. However, our transformation is applicable to this implementation, as we show next. (It seems likely that the argument in [10] can be adapted to show that it impossible to have a strongly-linearizable implementation of a multi-reader register using single-reader registers, as it is easy to simulate a message-passing channel with a single-reader register.)

In the implementation, a single-reader register *Val[i]* is associated with each reader $i$ of the implemented register. To **Write** a value $v$ to the implemented register, the (unique) writer writes $v$, together with a sequence number, into all of the *Val* registers. The readers communicate with each other via a (two-dimensional) array *Report* of single-reader registers, where reader $i$ writes to all the registers in row $i$ and reads from all the registers in column $i$. When a **Read** of the implemented register is invoked at process $i$, it reads (value, sequence number) pairs from *Val[i]* and from all the registers in column $i$ of *Report*; it then chooses the value to return with the largest sequence number, writes this pair to all the registers in row $i$ of *Report*, and returns. This implementation is tail strongly linearizable: the preamble of the **Read** method ends just before the first write to an element of *Report*, while the preamble of the **Write** method is empty. As before, the proof of tail strong linearizability is similar to the one for the ABD register.

# 6 RELATED WORK

Golab, Higham and Woelfel [14] were the first to recognize the problem when linearizable objects are used with randomized programs, via an example using the snapshot object implementation of [1]. They proposed *strong linearizability* as a way to overcome the increased vulnerability of programs using linearizable implementations to strong adversaries, by requiring that the linearization order of operations at any point in time be consistent with the

linearization order of each prefix of the execution. Thus, strongly-linearizable implementations limit the adversary's ability to gain additional power by manipulating the order of internal steps of different processes. Consequently, properties holding when a concurrent program is executed with an atomic object continue to hold when the program is executed with a strongly-linearizable implementation of the object. Strong linearizability is a special case of our class of implementations, where the preamble of each operation is empty and thus, vacuously, effect-free; in this case, applying the preamble-iterating transformation results in no change to the implementation.

Other than [6, 10] which studied message-passing implementations, prior work on strong linearizability focused on implementations using shared objects, and considered various progress properties. If one only needs *obstruction-freedom*, which requires an operation to complete only if it executes alone, any object can be implemented using single-writer registers [19]. When considering the stronger property of *lock-freedom* (or *nonblocking*), which requires that as long as some operation is pending, some operation completes, single-writer registers are not sufficient for implementing multi-writer registers, max registers, snapshots, or counters [19]. If the implementations can use multi-writer registers, though, it is possible to get lock-free implementations of max registers, snapshots, and monotonic counters [12], as well as of objects whose operations commute or overwrite [23]. It was also shown [4] that there is no lock-free implementation of a queue or a stack from objects whose readable versions have consensus number less than the number of processes, e.g., readable test&set. For the even stronger property of *wait-freedom*, which requires every operation to complete, it is possible to implement bounded max registers using multi-writer registers [19], but it is impossible to implement max registers, snapshots, or monotonic counters [12] even with multi-writer registers. The bottom line is that the only known strongly-linearizable wait-free implementation is of a bounded max register (using multi-writer registers), while many impossibility results are known.

*Write strong linearizability (WSL)* [18] is a weakening of strong linearizability designed specifically for register objects. It requires that executions be mapped to linearizations where only the projections onto write operations are prefix-preserving. While single-writer registers are trivially WSL, neither the original multi-writer ABD nor the preamble-iterating version we introduce in this paper is WSL [18]. The WSL implementation given in [18] has effect-free preambles, and so our transformation is applicable to it. It is not known whether it is possible to implement WSL multi-writer registers in crash-prone message-passing systems.

Our approach draws (loose) inspiration from the vast research on *oblivious RAM* (*ORAM*) (initiated in [15]), although the goals and technical details significantly differ. ORAMs provide an interface through which a program can hide its memory access pattern, while at the same time accessing the relevant information. More generally, *program obfuscation* [9] tries to hide (obfuscate) from an observer knowledge about the program's functionality, beyond what can be obtained from its input-output behavior. The goal of ORAMs and program obfuscation is to hide information from an adversary, while our goal is to blunt the adversary's ability to disrupt the program's behavior by exploiting linearizable implementations used by the program. We borrow, however, the key idea of introducing additional randomization into the implementation, in order to make it less vulnerable to the adversary.

## 7 DISCUSSION

We have presented the preamble-iterating transformation for a variety of linearizable object implementations, e.g., [1, 3, 21, 24], which approaches the probability of reaching particular outcomes, when these implementations replace the corresponding atomic objects. In this manner, it salvages randomized programs that use these highly-useful objects—which lack strongly-linearizable implementations—so they still terminate, without modifying the programs or their correctness proofs. Furthermore, the transformation is mechanical, once the preamble is identified.

Our results are just the first among many new opportunities for modular use of object libraries in randomized concurrent programs, including the following exciting avenues for future research.

One direction is to improve our analysis and obtain better bounds, specifically, by exploring the tradeoff between the increased complexity of many repetitions of the preamble, and decreased probability of bad outcomes.

Our transformation introduces computational overhead that depends on the number of program random steps that must be considered. Thus it is crucial to reduce, or at least bound, this number. This can be done by making assumptions about the structure of the randomized concurrent program. For example, many randomized programs are round-based, where each process takes a fixed (often, constant) number $s$ of random steps in each round, and termination occurs with high probability within some number of rounds, say $T$. In this case, we can let the program run for $T$ rounds and apply the preamble-iterating transformation with $k > T \cdot s$; if the program does not terminate within $T$ rounds, which happens with small probability, the program just continues with the original, linearizable object. An alternative approach for dealing with an unbounded number of random steps is to assume that the rounds are *communication-closed* [13], resulting in a smaller number of random choices that could affect the linearizable implementation. For example, a common style of algorithm partitions the code into phases (or rounds) and has each of $n$ processes flip a single coin in each phase, i.e., $r = n$. If our transformation is applied to each phase by repeating the preamble of each operation $k = 2r = 2n$ times, good probabilistic behavior is achieved with linear (in $n$) overhead. See [7, Section 2.4] for an example along these lines.

Another direction is to consider other objects without wait-free strongly-linearizable implementations, e.g., queues or stacks [4], which lack effect-free preambles that can be easily repeated. For such objects, it might be possible to *roll back* the effects of repeating certain parts of their implementation.

## ACKNOWLEDGMENTS

Hagit Attiya is partially supported by the Israel Science Foundation (grant number 380/18). Jennifer L. Welch is supported in part by the U.S. National Science Foundation under grant number 1816922.

## REFERENCES

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic Snapshots of Shared Memory. *J. ACM* 40, 4 (1993), 873–890.

[2] James Aspnes. 2003. Randomized protocols for asynchronous consensus. *Distributed Computing* 16, 2-3 (2003), 165–175.

[3] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-passing Systems. *J. ACM* 42, 1 (1995), 124–142.

[4] Hagit Attiya, Armando Castañeda, and Danny Hendler. 2018. Nontrivial and universal helping for wait-free queues and stacks. *J. Parallel and Distrib. Comput.* 121 (2018), 1–14.

[5] Hagit Attiya and Constantin Enea. 2019. Putting Strong Linearizability in Context: Preserving Hyperproperties in Programs that Use Concurrent Objects. In *33rd International Symposium on Distributed Computing (DISC)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:17.

[6] Hagit Attiya, Constantin Enea, and Jennifer L. Welch. 2021. Impossibility of Strongly-Linearizable Message-Passing Objects via Simulation by Single-Writer Registers. In *35th International Symposium on Distributed Computing (DISC)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:18.

[7] Hagit Attiya, Constantin Enea, and Jennifer L. Welch. 2021. Linearizable Implementations Suffice for Termination of Randomized Concurrent Programs (version 1). *CoRR* abs/2106.15554 (2021). https://arxiv.org/abs/2106.15554v1

[8] Hagit Attiya, Constantin Enea, and Jennifer L. Welch. 2022. Blunting an Adversary Against Randomized Concurrent Programs with Linearizable Implementations. *CoRR* abs/2106.15554 (2022). https://arxiv.org/abs/2106.15554

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

[10] David Yu Cheng Chan, Vassos Hadzilacos, Xing Hu, and Sam Toueg. 2021. An Impossibility Result on Strong Linearizability in Message-Passing Systems. *CoRR* abs/2108.01651 (2021). https://arxiv.org/abs/2108.01651

[11] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

[12] Oksana Denysyuk and Philipp Woelfel. 2015. Wait-Freedom is Harder Than Lock-Freedom Under Strong Linearizability. In *Distributed Computing - 29th International Symposium (DISC)*. Springer, 60–74.

[13] Tzilla Elrad and Nissim Francez. 1982. Decomposition of Distributed Programs into Communication-Closed Layers. *Sci. Comput. Program.* 2, 3 (1982), 155–173. https://doi.org/10.1016/0167-6423(83)90013-8

[14] Wojciech Golab, Lisa Higham, and Philipp Woelfel. 2011. Linearizable Implementations Do Not Suffice for Randomized Distributed Computation. In *Proceedings of the 43rd ACM Symposium on Theory of Computing (STOC)*. ACM, New York, NY, USA, 373–382.

[15] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473.

[16] Vassos Hadzilacos, Xing Hu, and Sam Toueg. 2020. On Atomic Registers and Randomized Consensus in M&M Systems (version 4). *CoRR* abs/1906.00298 (2020). http://arxiv.org/abs/1906.00298

[17] Vassos Hadzilacos, Xing Hu, and Sam Toueg. 2020. On Linearizability and the Termination of Randomized Algorithms. *CoRR* abs/2010.15210 (2020). http://arxiv.org/abs/2010.15210

[18] Vassos Hadzilacos, Xing Hu, and Sam Toueg. 2021. On Register Linearizability and Termination. In *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 521–531.

[19] Maryam Helmi, Lisa Higham, and Philipp Woelfel. 2012. Strongly linearizable implementations: possibilities and impossibilities. In *ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, New York, NY, USA, 385–394.

[20] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492.

[21] Amos Israeli and Ming Li. 1993. Bounded Time-Stamps. *Distributed Computing* 6, 4 (1993), 205–209.

[22] Nancy A Lynch and Alexander A Shvartsman. 1997. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. IEEE, 272–281.

[23] Sean Ovens and Philipp Woelfel. 2019. Strongly Linearizable Implementations of Snapshots and Other Types. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, (PODC)*. ACM, New York, NY, USA, 197–206.

[24] Paul M. B. Vitányi and Baruch Awerbuch. 1986. Atomic Shared Register Access by Asynchronous Hardware. In *27th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 233–243.