Non-determinism in Byzantine Fault-Tolerant Replication

Christian Cachin, Simon Schubert, and Marko Vukolić

IBM Research - Zurich, (cca|sis|mvu)@zurich.ibm.com

Abstract

Service replication distributes an application over many processes for tolerating faults, attacks, and misbehavior among a subset of the processes. With the recent interest in blockchain technologies, distributed execution of one logical application has become a prominent topic. The established state-machine replication paradigm inherently requires the application to be deterministic. This paper distinguishes three models for dealing with non-determinism in replicated services, where some processes are subject to faults and arbitrary behavior (so-called Byzantine faults): first, the modular case that does not require any changes to the potentially non-deterministic application (and neither access to its internal data); second, master-slave solutions, where ties are broken by a leader and the other processes validate the choices of the leader; and finally, applications that use cryptography and secret keys. Cryptographic operations and secrets must be treated specially because they require strong randomness to satisfy their goals.

The paper also introduces two new protocols. First, Protocol *Sieve* uses the modular approach and filters out non-deterministic operations in an application. It ensures that all correct processes produce the same outputs and that their internal states do not diverge. A second protocol, called *Mastercrypt*, implements cryptographically secure randomness generation with a verifiable random function and is appropriate for most situations in which cryptographic secrets are involved. All protocols are described in a generic way and do not assume a particular implementation of the underlying consensus primitive.

1998 ACM Subject Classification C.2.4 Distributed Systems; D.1.3 Concurrent Programming.

Keywords and phrases Blockchain, atomic broadcast, consensus, distributed cryptography, verifiable random functions

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

State-machine replication is an established way to enhance the resilience of a client-server application [33]. It works by executing the service on multiple independent components that will not exhibit correlated failures. We consider the approach of *Byzantine fault-tolerance (BFT)*, where a group of *processes* connected only by an unreliable network executes an application [31]. The processes use a protocol for *consensus* or *atomic broadcast* to agree on a common sequence of operations to execute. If all processes start from the same initial state, if all operations that modify the state are *deterministic*, and if all processes execute the same sequence of operations, then the states of the correct processes will remain the same. (This is also called *active* replication [13].) A client executes an operation on the service by sending the operation to all processes; it obtains the correct outcome based on comparing the responses that it receives, for example, by a relative majority among the answers or from a sufficiently large set of equal responses. Tolerating *Byzantine faults* means that the clients obtain correct outputs as long as a qualified majority of the processes is correct, even if the faulty processes behave in arbitrary and adversarial ways.

Traditionally state-machine replication requires the application to be deterministic. But many applications contain implicit or explicit non-determinism: in multi-threaded applications, the scheduler may influence the execution, input/output operations might yield different results across the processes, probabilistic algorithms may access a random-number generator, and some cryptographic operations are inherently not deterministic.

Recently BFT replication has gained prominence because it may implement distributed consensus for building *blockchains* [1, 15, 35, 5]. A blockchain provides a distributed, append-only ledger with cryptographic verifiability and is governed by decentralized control. It can be used to record events, trades, or transactions immutably and permanently and forms the basis for cryptocurrencies, such as Bitcoin or Ripple, or for running "smart contracts," as in Ethereum. With the focus on active replication, this work aims at *permissioned* blockchains, which run among known entities [34]. In contrast, *permissionless* blockchains (including Ethereum) do not rely on identities and use other approaches for reaching consensus, such as proof-of-work protocols. For practical use of blockchains, ensuring deterministic operations is crucial since even the smallest divergence among the outputs of different participants lets the blockchain diverge (or "fork").

This work presents a *general treatment* of non-determinism in the context of BFT replication and introduces a distinction among different models to tackle the problem of non-determinism. For example, applications involving cryptography and secret encryption keys should be treated differently from those that access randomness for other goals. We also distinguish whether the replication mechanism has access to the application's source code and may modify it.

We also introduce two novel protocols. The first, called *Sieve*, replicates non-deterministic programs using in a *modular* way, where we treat the application as a black box and cannot change it. We target workloads that are usually deterministic, but which may occasionally yield diverging outputs. The protocol initially executes all operations speculatively and then compares the outputs across the processes. If the protocol detects a minor divergence among a small number of processes, then we *sieve out the diverging values*; if a divergence among too many processes occurs, we *sieve out the operation* from sequence. Furthermore, the protocol can use *any* underlying consensus primitive to agree on an ordering. The second new protocol, *Mastercrypt*, provides master-slave replication with cryptographic security from verifiable random functions. It addresses situations that require strong, cryptographically secure randomness, but where the faulty processes may leak their secrets.

1.1 Contributions

We introduce three different models and discuss corresponding protocols for replicating non-deterministic applications.

Modular: When the application itself is fixed and cannot be changed, then we need *modular* replicated execution. In practice this is often the case. We distinguish two approaches for integrating a consensus protocol for ordering operations with the replicated execution of operations. One can either use *order-then-execute*, where the operations are ordered first, executed independently, and the results are communicated to the other processes through atomic broadcast. This involves only deterministic steps and can be viewed as "agreement on the input." Alternatively, with *execute-then-order*, the processes execute all operations speculatively first and then "agree on the output" (of the operation). In this case operations with diverging results may have to be rolled back.

We introduce Protocol *Sieve* that uses speculative execution and follows the *execute-then-order* approach. As described before, *Sieve* is intended for applications with occasional non-determinism. It represents the first modular solution to replicating non-deterministic applications in a BFT system.

Master-slave: In the *master-slave* model, one process is designated as the master or "leader," makes all non-deterministic choices that come up, and imposes these on the others which act as slaves or "followers." Because a faulty (Byzantine) master may misbehave, the slaves must be able to validate the selections of the master before the operation can be executed as determined by the master. The master-slave model is related to passive replication; it works for most applications including probabilistic algorithms, but cannot be applied directly for cryptographic operations. As a further complication, this model requires that the developer has access to the internals of the application and can modify it.

For the master-slave model we give a detailed description of the well-known replication protocol, which has been used in earlier systems.

Cryptographically secure: Traditionally, randomized applications can be made deterministic by deriving pseudorandom bits from a secret seed, which is initially chosen truly randomly. Outsiders, such as clients of the application, cannot distinguish this from an application that uses true randomness. This approach does not work for BFT replication, where faulty processes might expose and leak the seed. To solve this problem, we introduce a novel protocol for master-slave replication with cryptographic randomness, abbreviated *Mastercrypt*. It lets the master select random bits with a *verifiable random function*. The protocol is aimed at applications that need strong, cryptographically secure randomness; however it does not protect against a faulty master that leaks the secret. We also review the established approach of threshold (public-key) cryptography, where private keys are secret-shared among the processes and cryptographic operations are distributed in a fault-tolerant way over the whole group.

The modular Protocol *Sieve* has been developed for running potentially non-deterministic smart contracts as applications on top of a permissioned blockchain platform, built using BFT replication. An implementation has been made available as open source in "Hyperledger fabric" (`https://github.com/hyperledger/fabric`), which is part of the Linux Foundation's Hyperledger Project. As of November 2016, the project has decided to adopt a different architecture (`https://github.com/hyperledger/fabric/blob/master/docs/`); the platform has been redesigned to use a master-slave approach for addressing non-deterministic execution.

1.2 Related work

The problem of ensuring deterministic operations for replicated services is well-known. When considering only crash faults, many authors have investigated methods for making services deterministic, especially for multi-threaded, high-performance services [2]. Practical systems routinely solve this problem today using master-slave replication, where the master removes the ambiguity and sends deterministic updates to the slaves. In recent research on this topic, for instance, Kapitza et al. [24] present an optimistic solution for making multithreaded applications deterministic. Their solution requires a predictor for non-deterministic choices and may invoke additional communication via the consensus module.

In the BFT model, most works consider only sequential execution of deterministic commands, including PBFT [10] and UpRight [14]. BASE [11] and CBASE [26] address

Byzantine faults and adopt the master-slave model for handling non-determinism, focusing on being generic (BASE) and on achieving high throughput (CBASE), respectively. These systems involve changes to the application code and sometimes also need preprocessing steps for operations.

Fault-tolerant execution on multi-core servers poses a new challenge, even for deterministic applications, because thread-level parallelism may introduce unpredictable differences between processes. Eve [25] heuristically identifies groups of non-interfering operations and executes each group in parallel. Afterwards it compares the outputs, may roll back operations that lead to diverging states, or could transfer an agreed-on result state to diverging processes. Eve resembles Protocol *Sieve* in this sense, but lacks modularity.

For the same domain of scalable services running on multi-cores, Rex [21] uses the master-slave model, where the master executes the operations first and records its non-deterministic choices. The slaves replay these operations and use a consensus primitive to agree on a consistent outcome. Rex only tolerates crashes, but does not address the BFT model.

Fault-tolerant replication involving cryptographic secrets and distributed cryptography has been pioneered by Reiter and Birman [32]. Many other works followed, especially protocols using threshold cryptography; an early overview of solutions in this space was given by Cachin [4].

In current work Duan and Zhang [18] discuss how the master-slave approach can handle randomized operations in BFT replication, where execution is separated from agreement in order to protect the privacy of the data and computation.

1.3 Organization

The remainder of this paper starts with Section 2, containing background information and formal definitions of broadcast, replication, and atomic broadcast (i.e., consensus). The following sections contain the discussion and protocols for the three models: the modular solution (Section 3), the master-slave protocol (Section 4), and replication methods for applications demanding cryptographic security (Section 5). Due to lack of space in this extended abstract, many details, definitions, protocol optimizations, and discussions are only available in the full version [9].

2 Definitions

2.1 System model

We consider a distributed system of *processes* that communicate with each other and provide a common *service* in a fault-tolerant way. Using the paradigm of service replication [33], requests to the service are broadcast among the processes, such that the processes execute all requests in the same order. The clients accessing the service are not modeled here. We denote the set of processes by $\mathcal{P}$ and let $n = |\mathcal{P}|$. A process may be *faulty*, by crashing or by exhibiting *Byzantine faults*; the latter means they may deviate arbitrarily from their specification. Non-faulty processes are called *correct*. Up to f processes may be faulty and we assume that $n > 3f$. The setup is also called a *Byzantine fault-tolerant (BFT) service replication system* or simply a *BFT system*.

We present protocols in a modular way using an event-based notation [6]. A process is specified through its interface, consumes input events, and generates output events. Every two processes can *send* messages to each other using an authenticated point-to-point communication primitive. When a message arrives, the receiver learns also which process

has sent the message. The primitive guarantees *message integrity*, i.e., when a message m is received by a correct process with indicated sender p_s, and p_s is correct, then p_s previously sent m.

The system is *partially synchronous* [19] in the sense that there is no a priori bound on message delays and the processes have no synchronized clocks, as in an asynchronous system. However, there is a time (not known to the processes) after which the system is *stable* in the sense that message delays and processing times are bounded. In other words, the system is *eventually synchronous*. This model represents a broadly accepted network model and covers a wide range of real-world situations.

2.2 Broadcast and state-machine replication

Atomic broadcast. Suppose n processes participate in a broadcast primitive. Every process may *broadcast* a request or message m to the others. When a request has been agreed, it is *delivered*. Atomic broadcast also solves the *consensus* problem [22, 6]. We use a variant that delivers only messages satisfying a given *external validity* condition [7].

More precisely, *Byzantine atomic broadcast with external validity* (*abv*) is defined with a validation predicate $V()$ and uses two events: *abv-broadcast*(m), to broadcast a message m to all processes, and *abv-deliver*(p, m), which delivers a message m broadcast by process p.

Predicate $V()$ validates messages. It can be computed locally by every process and ensures that a correct process only delivers messages that satisfy $V()$. More precisely, $V()$ must guarantee that when two correct processes p and q have both delivered the same sequence of messages up to some point, then p obtains $V(m) = \textsc{True}$ for any message m if and only if q also determines that $V(m) = \textsc{True}$. The standard properties of Byzantine atomic broadcast [6] (validity, no duplication, integrity, agreement, and total order) are extended by:

External validity: When a correct process delivers some message m, then $V(m) = \textsc{True}$.

In practice it may occur that not all processes agree in the above sense on the validity of a message. For instance, some correct process may conclude $V(m) = \textsc{True}$ while others find that $V(m) = \textsc{False}$. For this case it is useful to reason with the following relaxation:

Weak external validity: When a correct process delivers some message m, then at least one correct process has determined that $V(m) = \textsc{True}$ at some time between when m was broadcast and when it was delivered.

Every protocol for Byzantine atomic broadcast with external validity of which we are aware either ensures this weaker notion or can easily be changed to satisfy it.

State machine replication. Atomic broadcast is the main tool to implement state-machine replication (SMR), which executes a service on multiple processes for tolerating process faults. Throughout this work we assume that many operation requests are generated concurrently by all processes; in other words, there is request contention.

A *state machine* consists of variables and operations that transform its state and may produce some output. Traditionally, operations are *deterministic*. The state machine *functionality* is defined by *execute*(), a function that takes a *state* $s \in \mathcal{S}$, initially s_0, and operation $o \in \mathcal{O}$ as input, and outputs a successor state s' and a *response* or *output value* r, such that *execute*$(s, o) \rightarrow (s', r)$.

A *replicated state-machine* is defined by two events: an input event *rsm-execute*(*operation*) that a process uses to invoke the execution of an operation o of the state machine; and an output event *rsm-output*(o, s, r), which is produced by the state machine. The output

indicates the operation has been executed and carries the resulting state s and response r. We assume here that an operation o includes both the name of the operation to be executed and any relevant parameters.

More formally, a *replicated state machine (rsm)* receives requests that the state machine executes the operation o, in the form of *rsm-execute*(o) events; it produces *rsm-output*(o, s, r) events, to indicate that the state machine has executed an operation o, resulting in new state s, and producing response r. It is defined using standard properties [6], ensuring *agreement* on the executed sequence of operations among all correct processes; *correctness* in the sense that when a correct process has executed a sequence of operations $o_1, \ldots, o_k$, then the sequences of output states $s_1, \ldots, s_k$ and responses $r_1, \ldots, r_k$ satisfies $(s_i, r_i) = execute(s_{i-1}, o_i)$ for $i = 1, \ldots, k$; and finally, *termination*.

The standard implementation of a replicated state machine relies on an atomic broadcast protocol to disseminate the requests to all processes [33, 22].

2.3 Leader election

Implementations of atomic broadcast need to make some synchrony assumptions or employ randomization [20]. A very weak timing assumption that is also available in many practical implementations is an *eventual leader-detector oracle* [12, 22].

We define an eventual leader-detector primitive, denoted Ω, for a system with Byzantine processes. It informs the processes about one correct process that can serve as a leader, so that the protocol can progress. When faults are limited to crashes, such a leader detector can be implemented from a failure detector [12], a primitive that, in practice, exploits timeouts and low-level point-to-point messages to determine whether a remote process is alive or has crashed.

With processes acting in arbitrary ways, though, one cannot rely on the timeliness of simple responses for detecting Byzantine faults. One needs another way to determine remotely whether a process is faulty or performs correctly as a leader. Detecting misbehavior in this model depends inherently on the specific protocol being executed [17]. We use the approach of "trust, but verify," where the processes monitor the leader for correct behavior. More precisely, a leader is chosen arbitrarily, but ensuring a fair distribution among all processes (in fact, it is only needed that a correct process is chosen at least with constant probability on average, over all leader changes). Once elected, the chosen leader process gets a chance to perform well. The other processes monitor its actions. Should the leader not have achieved the desired goal after some time, they complain against it, and initiate a switch to a new leader.

This notion of "performance" depends on the specific algorithm executed by the processes, which relies on the output from the leader-detection module. Therefore, eventual leader election with Byzantine processes is not an isolated low-level abstraction, as with crash-stop processes, but requires some input from the higher-level algorithm. The Ω-*complain*(p) event allows to express this. Every process may *complain* against the current leader p by triggering this event.

Formally, a *Byzantine leader detector* (Ω) is defined with an output Ω-*trust*(p), designating process p to be trusted as leader, and an input event Ω-*complain*(p) that receives a complaint about the performance of leader process p. Its formal properties [6] ensure that eventually, every correct process trusts some correct process; that when more than f correct processes that trust some process p complain about p, then every correct process eventually trusts a different process than p. Moreover, a correct process q does not trust a new leader unless at least one correct process has complained against the leader which q trusted before, and that

eventually no two correct processes trust different processes.

It is possible to lift the output from the Byzantine leader detector to an *epoch-change* primitive, which outputs not only the identity of a leader but also an increasing *epoch number*. This abstraction divides time into a series of epochs at every participating process, where epochs are identified by numbers. The numbers of the epochs started by one particular process increase monotonically (but they do not have to form a complete sequence). Moreover, the primitive also assigns a *leader* to every epoch, such that any two correct processes in the same epoch receive the same leader. The mechanism for processes to complain about the leader is the same as for Ω.

More precisely, *Byzantine epoch-change* (Ψ) outputs events of the form Ψ-*start-epoch*(e, p), which indicate that epoch with number e and leader p starts; it also receives Ψ-*complain*(e, p) events similar to Ω. Its formal properties appears in the literature [6].

When an epoch-change abstraction is initialized, it is assumed that a default epoch with number 0 and a leader p_0 has been started at all correct processes. All "practical" BFT systems in the eventual-synchrony model starting from PBFT [10] implicitly contain an implementation of Byzantine epoch-change; this notion was described explicitly by Cachin et al. [6, Chap. 5].

3 Modular protocol

In this section we discuss the *modular* execution of replicated non-deterministic programs. Here the program is given as a black box, it cannot be changed, and the BFT system cannot access its internal data structures. Very informally speaking, if some processes arrive at a different output during execution than "most" others, then the output of the disagreeing processes is discarded. Instead they should "adopt" the output of the others, e.g., by asking them for the agreed-on state and response. When the outputs of "too many" processes disagree, the correct output may not be clear; the operation is then ignored (or, as an optimization, quarantined as non-deterministic) and the state rolled back. In this modular solution any application can be replicated without change; the application developers may not even be aware of potential non-determinism. On the other hand, the modular protocol requires that most operations are deterministic and produce almost always the same outputs at all processes; it would not work for replicating probabilistic functions.

More precisely, a *non-deterministic state machine* may output different states and responses for the same operation, which are due to probabilistic choices or other non-repeatable effects. Hence we assume that *execute* is a relation and not a deterministic function, that is, repeated invocations of the same operation with the same input may yield different outputs and responses. This means that the standard approach of state-machine replication based directly on atomic broadcast fails.

There are two ways for modular black-box replication of non-deterministic applications in a BFT system:

Order-then-execute: Applying the SMR principle directly, the operations are first ordered by atomic broadcast. Whenever a process delivers an operation according to the total order, it executes the operation. It does not output the response, however, before checking with enough others that they all arrive at the same outputs. To this end, every process atomically broadcasts its outputs (or a hash of the outputs) and waits for receiving a given number (up to $n - f$) of outputs from distinct processes. Then the process applies a fixed decision function to the atomically delivered outputs, and it determines the successor state and the response.

This approach ensures consistency due to its conceptual simplicity but is not very efficient in typical situations, where atomic broadcast forms the bottleneck. In particular, in atomic broadcast with external validity, a process can only participate in the ordering of the next operation when it has determined the outputs of the previous one. This eliminates potential gains from pipelining and increases the overall latency.

Execute-then-order: Here the steps are inverted and the operations are executed *speculatively* before the system commits their order. As in other practical protocols, this solution uses the heuristic assumption that there is a designated *leader* which is usually correct. Thus, every process sends its operations to the leader and the leader orders them. It asks all processes to execute the operations speculatively in this order, the processes send (a hash of) their outputs to the leader, and the leader determines a unique output. Note that this value is still speculative because the leader might fail or there might be multiple leaders acting concurrently. The leader then tries to obtain a confirmation of its speculative order by atomically broadcasting the chosen output. Once every process obtains this output from atomic broadcast, it commits the speculative state and outputs the response.

In rare cases when a leader is replaced, some processes may have speculated wrongly and executed other operations than those determined through atomic broadcast. Due to non-determinism in the execution a process may also have obtained a different speculative state and response than what the leader has obtained and broadcast. This implies that the leader must either send the state (or state delta) and the response resulting from the operation though atomic broadcast, or that a process has a different way to recover the decided state from other processes.

In the following we describe Protocol *Sieve*, which adopts the approach of *execute-then-order* with speculative execution.

Protocol *Sieve*. Protocol *Sieve* runs a Byzantine atomic broadcast with weak external validity (abv) and uses a *sieve-leader* to coordinate the execution of non-deterministic operations. The leader is elected through a Byzantine epoch-change abstraction, as defined in Section 2.3, which outputs epoch/leader tuples with monotonically increasing epoch numbers. For the *Sieve* protocol these epochs are called *configurations*, and *Sieve* progresses through a series of them, each with its own sieve-leader.

The processes send all operations to the service through the leader of the current configuration, using an INVOKE message. The current leader then initiates that all processes execute the operation speculatively; subsequently the processes agree on an output from the operation and thereby *commit* the operation. As described here, *Sieve* executes one operation at a time, although it is possible to greatly increase the throughput using the standard method of *batching* multiple operations together.

The leader sends an EXECUTE message to all processes with the operation o. In turn, every process executes o *speculatively* on its current state s, obtains the speculative next state t and the speculative response r, signs those values, and sends a hash and the signature back to the leader in an APPROVE message.

The leader receives $2f + 1$ APPROVE messages from distinct processes. If the leader observes at least $f + 1$ approvals for the *same* speculative output, then it *confirms* the operation and proceeds to committing and executing it. Otherwise, the leader concludes that the operation is *aborted* because of diverging outputs. There must be $f + 1$ equal outputs for confirming o, in order to ensure that every process will eventually learn the correct output, see below.

The leader then *abv-broadcasts* an ORDER message, containing the operation, the speculative output (t, r) for a confirmed operation or an indication that it aborted, and for validation the set of APPROVE messages that justify the decision whether to confirm or abort. During atomic broadcast, the external validity check by the processes will verify this justification.

As soon as an ORDER message with operation o is *abv-delivered* to a process in *Sieve*, o is committed. If o is confirmed, the process adopts the output decided by the leader. Note this may differ from the speculative output computed by the process. Protocol *Sieve* therefore includes the next state t and the response r in the ORDER message. In practice, however, one might not send t, but state deltas, or even only the hash value of t while relying on a different way to recover the confirmed state. Indeed, since $f + 1$ processes have approved any confirmed output, a process with a wrong speculative output is sure to reach at least one of them for obtaining the confirmed output later.

In case the leader *abv-broadcasted* an ORDER message with the decision to abort the current operation because of the diverging outputs (i.e., no $f + 1$ identical hashes in $2f + 1$ APPROVE messages), the process simply ignores the current request and speculative state. As an optimization, processes may *quarantine* the current request and flag it as non-deterministic.

As described so far, the protocol is open to a denial-of-service attack by multiple faulty processes disguising as sieve-leaders and executing different operations. Note that the epoch-change abstraction, in periods of asynchrony, will not ensure that any two correct processes agree on the leader, as some processes might skip configurations. Therefore *Sieve* also orders the configuration and leader changes using consensus (with the *abv* primitive).

To this effect, whenever a process receives a *start-epoch* event with itself as leader, the process *abv-broadcasts* a NEW-SIEVE-CONFIG message, announcing itself as the leader. The validation predicate for broadcast verifies that the leader announcement concerns a configuration that is not newer than the most recently started epoch at the validating process, and that the process itself endorses the same next leader. Every process then starts the new configuration when the NEW-SIEVE-CONFIG message is *abv-delivered*. If there was a speculatively executed operation, it is aborted and its output discarded.

The design of *Sieve* prevents uncoordinated speculative request execution, which may cause contention among requests from different self-proclaimed leaders and can prevent liveness easily. Naturally, a faulty leader may also violate liveness, but this is not different from other leader-based BFT protocols.

The details of Protocol *Sieve* are shown in Algorithms 1–2. The pseudocode assumes that all point-to-point messages among correct processes are authenticated, cannot be forged or altered, and respect FIFO order. The invoked operations are unique across all processes and *self* denotes the identifier of the executing process. Not shown in the pseudocode is a periodic concurrent check for leader progress. The process determines the age of every $o \in \mathcal{I}$ since it has been invoked and added to $\mathcal{I}$; if there are "old" operations in $\mathcal{I}$, then the process invokes Ψ-*complain*(*leader*).

The following two optimizations for *Sieve* are described in the full version [9]: First, when run in practice, every process directly executes operations and does not include the potentially large state in ORDER messages. If a *rollback* operation exists to complement *execute*, a process that has computed a diverging state can roll the operation back and obtain the state from other processes. Second, when the well-known *PBFT protocol* [10] implements *abv-broadcast*, then the leader information and Byzantine epoch-change mechanism can be directly obtained from PBFT. This simplifies the description of *Sieve* but breaks modularity.

Algorithm 1 Protocol *Sieve*

State

$\mathcal{I}$: set of invoked operations at every process — $B[p]$, for $p \in \mathcal{P}$: buffer at sieve-leader
config: sieve-config number — *leader*: sieve-leader, initially p_0
next-epoch: next sieve-config, initially $\perp$ — *next-leader*: next sieve-leader, initially $\perp$
s: current state, initially s_0 — *cur*: current operation, initially $\perp$
t: speculative state, initially $\perp$ — r: speculative response, initially $\perp$

upon invocation *rsm-execute*(o) **do**
  $\mathcal{I} \leftarrow \mathcal{I} \cup \{o\}$
  send msg. [INVOKE, *config*, o] over point-to-point link to *leader*

upon recv. msg. [INVOKE, c, o] from p **such that** $B[p] = \perp$ **and** $c =$ *config* **and** *leader* $=$ *self* **do**
  $B[p] \leftarrow o$ // buffer only the latest operation from each process

upon exists p that $B[p] \neq \perp$ **such that** *cur* $= \perp$ **and** *leader* $=$ *self* **do**
  cur $\leftarrow B[p]$
  send [EXECUTE, *config*, *cur*] over point-to-point links to all processes

upon recv. msg. [EXECUTE, c, o] from p **such that** $p =$ *leader* **and** $c =$ *config* **and** $t = \perp$ **do**
  $(t, r) \leftarrow$ *execute*(s, o)
  $\sigma \leftarrow$ *sign*$_{self}$(SPECULATE$\|$*config*$\|$*hash*$(t\|r)$)
  send msg. [APPROVE, *config*, o, *hash*$(t\|r), \sigma$] to *leader*

upon recv. $2f + 1$ msgs. [APPROVE, c_p, o_p, h_p, σ_p], each from a distinct process p, **such that**
    $c_p =$ *config* **and** $op_p =$ *cur* **and** *verify*$_p(\sigma_p$, SPECULATE$\|$*config*$\|h_p)$ **and** *leader* $=$ *self* **do**
  if there is a set $\mathcal{E}$ of $f + 1$ received APPROVE msgs. whose h_p value is equal to *hash*$(t\|r)$ **then**
    abv-broadcast([ORDER, CONFIRM, *config*, *cur*, $t, r, \mathcal{E}$])
  else
    let $\mathcal{U}$ be the set of $2f + 1$ received APPROVE msgs.
    abv-broadcast([ORDER, ABORT, *config*, *cur*, $\perp, \perp, \mathcal{U}$])

upon *abv-deliver*(p, [ORDER, *decision*, $c, o, t_c, r_c, \cdot$]) **such that** $c =$ *config* **do** // commit o
  if *leader* $=$ *self* **then**
    $B[p] \leftarrow \perp$
    cur $\leftarrow \perp$
  if $o \in \mathcal{I}$ **then**
    $\mathcal{I} \leftarrow \mathcal{I} \setminus \{o\}$
  if *decision* $=$ CONFIRM **then**
    $s \leftarrow t_c$ // adopt the agreed-on state and response, needed if $(t_c, r_c) \neq (t, r)$
    rsm-output(o, s, r_c)
  $t \leftarrow \perp$

upon Ψ-*start-epoch*(e, p) **do**
  $($*next-epoch*, *next-leader*$) \leftarrow (e, p)$
  if $p =$ *self* $\wedge\ e >$ *config* **then**
    abv-broadcast([NEW-SIEVE-CONFIG, e, *self*])

upon *abv-deliver*(p, [NEW-SIEVE-CONFIG, c, p]) **do**
  $($*config*, *leader*$) \leftarrow (c, p)$
  $t \leftarrow \perp$

Algorithm 2 Validation predicate $V()$ for Byzantine atomic broadcast used inside Algorithm *Sieve*

upon invocation $V(m)$ **do**
  if $m = [\text{ORDER}, \text{DECISION}, c, o, \mathcal{M}]$ **then**
    if $\mathcal{M}$ is a set of $f+1$ msgs. of the form $[\text{APPROVE}, c_p, o_p, h_p, \sigma_p]$ **such that**
      $c_p = \mathit{config}$ **and** $o_p = o$ **and** $\mathit{verify}_p(\sigma_p, \text{SPECULATE}\|c_p\|h_p) = \text{TRUE}$ **and**
      all h_p values in $\mathcal{M}$ are equal **then**
    return TRUE
  else if $m = [\text{ORDER}, \text{ABORT}, c, o, \mathcal{M}]$ **then**
    if $\mathcal{M}$ is a set of $2f+1$ msgs. of the form $[\text{APPROVE}, c_p, o_p, h_p, \sigma_p]$ **such that**
      $c_p = \mathit{config}$ **and** $o_p = o$ **and** $\mathit{verify}_p(\sigma_p, \text{SPECULATE}\|c_p\|h_p) = \text{TRUE}$ **and**
      no $f+1$ of the h_p values in $\mathcal{M}$ are equal **then**
    return TRUE
  else if $m = [\text{NEW-SIEVE-CONFIG}, c, p]$ **then**
    if $c \leq \mathit{next\text{-}epoch}$ **and** $p = \mathit{next\text{-}leader}$ **then**
      return TRUE
  return FALSE

▶ **Theorem 1.** *Protocol* Sieve *implements a replicated state machine allowing a non-deterministic functionality* execute()*, except that demonstrably non-deterministic operations may be filtered out and not executed.*

To see why this holds, we consider first the *agreement* condition of a replicated state machine: this follows directly from the protocol and from the *abv* primitive. Every *rsm-output* event is immediately preceded by an *abv-delivered* ORDER message, which is the same for all correct processes due to *agreement* of *abv*. Since all correct processes react to it deterministically, their outputs are the same.

For the *correctness* property, note that the outputs (s_i, r_i) (state and response) resulting from an operation o must have been confirmed by the protocol and therefore the values were included in an APPROVE message from at least one correct process. This process computed the values such that they satisfy $(s_i, r_i) = \mathit{execute}(s_{i-1}, o)$ according to the protocol for handling an EXECUTE message. On the other hand, no correct process outputs anything for committed operations that were aborted, this is permitted by the exception in the theorem statement. Moreover, only operations are filtered out for which distinct correct processes computed diverging outputs, as ensured by the sieve-leader when it determines whether the operation is confirmed or aborted. In order to abort, no set of $f+1$ processes must have computed the same outputs among the $2f+1$ processes sending the APPROVE messages. Hence, at least two among every set of $f+1$ correct processes arrived at diverging outputs.

Termination is only required for deterministic operations, they must terminate despite faulty processes that approve wrong outputs. The protocol ensures this through the condition that at least $f+1$ among the $2f+1$ APPROVE messages received by the sieve-leader are equal. The faulty processes, of which there are at most f, cannot cause an abort through this. But every ORDER message is eventually *abv-delivered* and every confirmed operation is eventually executed and generates an output.

Discussion. Non-deterministic operations have not often been discussed in the context of BFT systems. The literature commonly assumes that deterministic behavior can be imposed on an application or postulates to change the application code for isolating non-determinism. In practice, however, it is often not possible.

Liskov [27] sketches an approach to deal with non-determinism in PBFT which is similar to *Sieve* in the sense that it treats the application code modularly and uses execute-then-order. This proposal is restricted to the particular structure of PBFT, however, and does not consider the notion of external validity for *abv* broadcast.

For applications on multi-core servers, the *Eve* system [25] also executes operation groups speculatively across processes and detects diverging states during a subsequent verification stage. In case of divergence, the processes must roll back the operations. The approach taken in Eve resembles that of *Sieve*, but there are notable differences. Specifically, the primary application of Eve continues to assume deterministic operations, and non-determinism may only result from concurrency during parallel execution of requests. Furthermore, this work uses a particular agreement protocol based on PBFT and not a generic *abv* broadcast primitive.

It should be noted that *Sieve* not only works with Byzantine atomic broadcast in the model of eventual synchrony, but can equally well be run over randomized Byzantine consensus [7, 30].

4 Master-slave protocol

By adopting the *master-slave* model one can support a broader range of non-deterministic application behavior compared to the modular protocol. This design generally requires source-code access and modifications to the program implementing the functionality. In a master-slave protocol for non-deterministic execution, one process is designated as *master*. The master executes every operation first and records all non-deterministic choices. All other processes act as *slaves* and follow the same choices. To cope with a potentially Byzantine master, the slaves must be given means to verify that the choices made by the master are plausible. The master-slave solution presented here follows *primary-backup replication* [3], which is well-known to handle non-deterministic operations. For instance, if the application accesses a pseudorandom number generator, only the master obtains the random bits from the generator and the slaves adopt the bits chosen by the master. This protocol does not work for functionalities involving cryptography, however, where master-slave replication typically falls short of achieving the desired goals. Instead a cryptographically secure protocol should be used; they are the subject of Section 5.

Non-deterministic execution with evidence. As introduced in Section 3, the *execute* operation of a non-deterministic state machine is a relation. Different output values are possible and represent acceptable outcomes. We augment the output of an operation execution by adding *evidence* for justifying the resulting state and response. The slave processes may then *replay* the choices of the master and accept its output.

More formally, we now extend *execute* to *nondet-execute* as follows:

$$\textit{nondet-execute}(s, o) \;\rightarrow\; (s', r, \rho).$$

Its parameters s, o, s', and r are the same as for *execute*; additionally, the function also outputs *evidence* ρ. Evidence enables the slave processes to execute the operation by themselves and obtain the same output as the master, or perhaps only to validate the output generated by another execution. For this task there is a function

$$\textit{verify-execution}(s, o, s', r, \rho) \;\rightarrow\; \{\textsc{False}, \textsc{True}\}$$

that outputs TRUE if and only if the set of possible outputs from *nondet-execute*(s, o) contains (s', r, ρ). For completeness we require that for every s and o, when $(s', r, \rho) \leftarrow$ *nondet-execute*(s, o), it always holds *verify-execute*$(s, o, s', r, \rho) =$ TRUE.

As a basic verification method, a slave could rerun the computation of the master. Extensions to use cryptographic verifiable computation [36] are possible. Note that we consider randomized algorithms to be a special case of non-deterministic ones. The evidence for executing a randomized algorithm might simply consist of the random coin flips made during the execution.

Replication protocol. Implementing a replicated state machine with non-deterministic operations using master-slave replication does not require an extra round of messages to be exchanged, as in Protocol *Sieve*. It suffices that the master is chosen by a Byzantine epoch-change abstraction and that the master broadcasts every operation together with the corresponding evidence.

More precisely, the processes operate on top of an underlying broadcast primitive *abv* and a Byzantine epoch-change abstraction Ψ. Whenever a process receives a *start-epoch* event with itself as leader from Ψ, the process considers itself to be the master for the epoch and *abv-broadcasts* a message that announces itself as the master for the epoch. The epochs evolve analogously to the configurations in *Sieve*, with the same mechanism to approve changes of the master in the validation predicate of atomic broadcast. Similarly, non-master processes send their operations to the master of the current epoch for ordering and execution.

For every invoked operation o, the master computes $(s', r, \rho) \leftarrow$ *nondet-execute*(s, o) and *abv-broadcasts* an ORDER message containing the current epoch c and parameters o, s', r, and ρ. The validation predicate of atomic broadcast for ORDER messages verifies that the message concerns the current epoch and that *verify-execution*$(s, o, s', r, \rho) =$ TRUE using the current state s of the process. Once an ORDER message is *abv-delivered*, a process adopts the response and output state from the message as its own.

Discussion. The master-slave protocol is inspired by primary-backup replication [3], and for the concrete scenario of a BFT system, it was first described by Castro, Rodrigues, and Liskov in BASE [11]. The protocol of BASE addresses only the particular context of PBFT, however, and not a generic atomic broadcast primitive. As mentioned before, the master-slave protocol requires changes to the application for extracting the evidence that will convince the slave processes that choices made by the master are valid.

5 Cryptographically secure protocols

Security functions implemented with cryptography are more important today than ever. Replicating an application that involves a cryptographic secret, however, requires a careful consideration of the attack model. If the BFT system should tolerate that f processes become faulty in arbitrary ways, it must be assumed that their secrets leak to the adversary against whom the cryptographic scheme is employed.

Service-level secret keys must be protected and should never leak to an individual process. Two solutions have been explored to address this issue. One could delegate this responsibility to a third party, such as a centralized service or a secure hardware module at every process. However, this contradicts the main motivation behind replication: to eliminate central control points. Alternatively one may use *distributed cryptography* [16], share the keys among the processes so that no coalition of up to f among them learns anything, and perform the

cryptographic operations under distributed control. This model was pioneered by Reiter and Birman [32] and exploited, for instance, by SINTRA [4, 8] or COCA [37].

In this section we introduce a novel protocol, called *Mastercrypt*, for integrating non-deterministic cryptographic operations in a BFT system, based on the master-slave paradigm and using verifiable random functions to generate pseudorandom bits. This randomness is unpredictable and cannot be biased by a Byzantine process. In the full version [9] we furthermore review a protocol based on the well-known idea of using distributed cryptography, as discussed above. Both schemes adopt the master-slave replication protocol from the previous section.

Randomness from verifiable random functions. A *verifiable random function (VRF)* [29] resembles a pseudorandom function but additionally permits anyone to verify non-interactively that the choice of random bits occurred correctly. The function therefore guarantees correctness for its output without disclosing anything about the secret seed, in a way similar to non-interactive zero-knowledge proofs of correctness.

Efficient implementations of VRFs have not been easy to find, but the literature nowadays contains a number of reasonable constructions under broadly accepted hardness assumptions [28, 23]. In practice, when adopting the random-oracle model, VRFs can immediately be obtained from unique signatures such as ordinary RSA signatures [28].

Protocol *Mastercrypt*: Replication with cryptographic randomness from a VRF. With master-slave replication, cryptographically strong randomness secure against faulty non-leader processes can be obtained from a VRF as follows. Initially every process generates a VRF-seed and a verification key. Then it passes the verification key to a trusted entity, which distributes the n verification keys to all processes consistently, ensuring that all correct processes use the same list of verification keys. At every place where the application needs to generate (pseudo-)randomness, the VRF is used by the master to produce the random bits and all processes verify that the bits are unique. Details of this protocol can be found in the full version [9].

6 Conclusion

This paper has introduced a distinction between three models for dealing with non-deterministic operations in BFT replication: *modular* where the application is a black box; *master-slave* that needs internal access to the application; and *cryptographically secure* handling of non-deterministic randomness generation. In the past, dedicated BFT replication systems have often argued for using the master-slave model, but we have learned in the context of blockchain applications that changes of the code and understanding an application's logic can be difficult. Hence, our novel Protocol *Sieve* provides a modular solution that does not require any manual intervention. For a BFT-based blockchain platform, *Sieve* can simply be run without incurring large overhead as a defense against non-determinism, which may be hidden in smart contracts.

Acknowledgments

We thank our colleagues and the members of the IBM Blockchain development team for interesting discussions and valuable comments, in particular Elli Androulaki, Konstantinos Christidis, Angelo De Caro, Chet Murthy, Binh Nguyen, and Michael Osborne.

This work was supported in part by the European Union's Horizon 2020 Framework Programme under grant agreement number 643964 (SUPERCLOUD) and in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0091.

References

1. J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In *Proc. 36th IEEE Symposium on Security & Privacy*, pages 104–121, 2015.
2. T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, Feb. 1996.
3. N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In *Distributed Systems (2nd Ed.)*. ACM Press & Addison-Wesley, New York, 1993.
4. C. Cachin. Distributing trust on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 183–192, 2001.
5. C. Cachin, editor. *Distributed Cryptocurrencies and Consensus Ledgers (DCCL 2016)*, Online proceedings of workshop co-located with PODC, 2016. `https://www.zurich.ibm.com/dccl/`.
6. C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.
7. C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols (extended abstract). In *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001.
8. C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 167–176, June 2002.
9. C. Cachin, S. Schubert, and M. Vukolić. Non-determinism in Byzantine fault-tolerant replication. e-print, arXiv:1603.07351 [cs.DC], 2016. URL: `http://arxiv.org/abs/1603.07351`.
10. M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.
11. M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, 21(3):236–269, 2003.
12. T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
13. B. Charron-Bost, F. Pedone, and A. Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.
14. A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 277–290, 2009.
15. K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, and R. Wattenhofer. On scaling decentralized blockchains. In *Proc. 3rd Workshop on Bitcoin and Blockchain Research*, 2016.
16. Y. Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–457, 1994.
17. A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *Proc. 3rd European Dependable Computing Conference (EDCC-3)*, volume 1667 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 1999.

18 S. Duan and H. Zhang. Practical confidential state machine replication: How to process data privately in the cloud. In *Proc. 35th Symposium on Reliable Distributed Systems (SRDS)*, 2016.

19 C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

20 M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

21 Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proc. 9th European Conference on Computer Systems (EuroSys)*, 2014.

22 V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*. ACM Press & Addison-Wesley, New York, 1993.

23 T. Jager. Verifiable random functions from weaker assumptions. In *Proc. 12th Theory of Cryptography Conference (TCC 2015)*, volume 9015 of *Lecture Notes in Computer Science*, pages 121–143. Springer, 2015.

24 R. Kapitza, M. Schunter, C. Cachin, K. Stengel, and T. Distler. Storyboard: Optimistic deterministic multithreading. In *Proc. 6th Workshop on Hot Topics in System Dependability*, 2010.

25 M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Proc. 10th Symp. Operating Systems Design and Implementation (OSDI)*, 2012.

26 R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 575–584, June 2004.

27 B. Liskov. From viewstamped replication to Byzantine fault tolerance. In *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 121–149. Springer, 2010.

28 A. Lysyanskaya. Unique signatures and verifiable random functions from the DH-DDH separation. In *Advances in Cryptology: CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 597–612. Springer, 2002.

29 S. Micali, M. Rabin, and S. Vadhan. Verifiable random functions. In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 120–130, 1999.

30 A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of BFT protocols. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2016.

31 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, Apr. 1980.

32 M. K. Reiter and K. P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, May 1994.

33 F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

34 T. Swanson. Consensus-as-a-service: A brief report on the emergence of permissioned, distributed ledger systems. Report, Apr. 2015. URL: `http://www.ofnumbers.com/wp-content/uploads/2015/04/Permissioned-distributed-ledgers.pdf`.

35 M. Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *Open Problems in Network Security, Proc. IFIP WG 11.4 Workshop (iNetSec 2015)*, volume 9591 of *Lecture Notes in Computer Science*, pages 112–125. Springer, 2016.

36 M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2), Feb. 2015.

37 L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.