Non-determinism in Byzantine Fault-Tolerant Replication

Christian Cachin Simon Schubert Marko Vukolić

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

2016-03-23

Abstract

Service replication distributes an application over many processes for tolerating faults, attacks, and misbehavior among a subset of the processes. 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, a modular approach that does not require any changes to the potentially non-deterministic application (and neither access to its internal data); second, a master-slave approach, in which ties are broken by a leader and the other processes validate the choices of the leader; and finally, a treatment of 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. The first uses the modular approach for filtering 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. The second protocol implements cryptographically secure randomness generation with a verifiable random function and is appropriate for certain security models. All protocols are described in a generic way and do not assume a particular implementation of the underlying consensus primitive.

1 Introduction

State-machine replication is an established way to enhance the resilience of a client-server application [31]. 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 [29]. 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. A client executes an operation on the service by sending the operation to all processes; it determines 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, 13, 35]. 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.

This work presents a *general treatment* of non-determinism in the context of BFT replication and introduces a distinction among multiple approaches 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 a *modular* approach, 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 uses verifiable random functions for replication in situations that require cryptographically secure randomness, but where the faulty processes may leak their secrets.

1.1 Contributions

We introduce three different models and provide 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 ways 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* model. 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* design one process is designated as the "leader," it 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 works for most applications including probabilistic algorithms, but it 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 approach 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 cannot distinguish this from an application that uses true randomness everywhere. 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 master-slave protocol where the master selects random bits with a *verifiable random function*; it addresses many security goals but does not protect against a faulty master. Furthermore we 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.

Protocol *Sieve*, in the modular approach, has been developed for executing potentially non-deterministic "smart contracts" as applications for a blockchain built with BFT replication. An implementation is available as open-source code (`github.com/openblockchain`) and was contributed to Linux Foundation's Hyperledger Project (`www.hyperledger.org`).

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 the master-slave approach, where the master removes the ambiguity and sends deterministic updates to the slaves. In recent research on this topic, for instance, Kapitza et al. [22] 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 [32]. BASE [11] and CBASE [24] address Byzantine faults and exploit the master-slave approach for handling non-determinism, focusing on a generic approach (BASE) and on 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 [23] 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 [19] uses the master-slave approach, 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, following the master-slave approach. 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 [30]. Many other works followed, especially protocols using threshold cryptography; an early overview of solutions in this space was given by Cachin [4].

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 approach (Section 3), the master-slave protocol (Section 4), and replication methods for applications demanding cryptographic security (Section 5).

2 Definitions

2.1 System model

We consider an asynchronous 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 [31], 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 [5]. A process is specified through its *interface*, containing the events that it exposes to other processes, and through a set of *properties*, which define its behavior. A process may react to a received event by doing computation and triggering further events. The events of a process interface consist of *input events*, which the process receives from other processes, typically to invoke its services, and *output events*, through which the process delivers information or signals a condition to another process.

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. Authenticated communication can be implemented easily from an insecure communication channel by using a message-authentication code (MAC) [27], a symmetric cryptographic primitive that relies on a secret key shared by every pair of processes. These keys have been distributed by a trusted entity beforehand.

The system is *partially synchronous* [16] 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

Suppose n processes participate in a broadcast primitive. Every process may *broadcast* a request or message m to the others. The implementation generates events to output the requests when they have been agreed; we say that a request m is *delivered* through this. Atomic broadcast also solves the *consensus* problem [20, 5]. We use a variant that delivers only messages satisfying a given *external validity* condition [6].

Definition 1 (Byzantine atomic broadcast with external validity). A *Byzantine atomic broadcast with external validity* (*abv*) is defined with the help of a validation predicate $V()$ and in terms of these events:

– *Input event:* abv-broadcast(m): Broadcasts a message m to all processes.

– *Output event:* abv-deliver(p, m): Delivers a message m broadcast by process p.

The deterministic predicate $V()$ validates messages. It can be computed locally by every process. It 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 $V(m) = \text{TRUE}$ for any message m if and only if q also determines that $V(m) = \text{TRUE}$.

With this validity mechanism, the broadcast satisfies:

– *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

– *External validity:* When a correct process delivers some message m, then $V(m) = \text{TRUE}$.

– *No duplication:* No correct process delivers the same message more than once.

– *Integrity:* If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcast by p.

– *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

– *Total order:* Let m_1 and m_2 be any two messages and suppose p and q are any two correct processes that deliver m_1 and m_2. If p delivers m_1 before m_2, then q delivers m_1 before m_2.

In practice it may occur that not all processes agree in the above sense on the validity of a message. It may occur that some correct process concludes $V(m) = \text{TRUE}$ while others find that $V(m) = \text{FALSE}$. 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 $V(m) = \text{TRUE}$ 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.

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* and the outputs of the state machine are solely determined by the initial state and by the sequence of operations that it has executed.

The state machine *functionality* is defined by $\mathit{execute}()$, an operation 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:

$$\mathit{execute}(s, o) \;\rightarrow\; (s', r),$$

A *replicated* state-machine primitive can be characterized as in Definition 2. Basically, its interface presents two events: first, an input event $\mathit{rsm\text{-}execute}(\mathit{operation})$ that a process uses to invoke the execution of an operation o of the state machine; and second, an output event $\mathit{rsm\text{-}output}(o, s, r)$, 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.

Definition 2. A *replicated state machine (rsm)* for a functionality $\mathit{execute}()$ and initial state s_0 is defined by these events:

– *Input event:* $\mathit{rsm\text{-}execute}(o)$: Requests that the state machine executes the operation o.

– *Output event:* $\mathit{rsm\text{-}output}(o, s, r)$: Indicates that the state machine has executed an operation o, resulting in new state s, and producing response r.

It also satisfies these properties:

– *Agreement:* The sequences of executed operations and corresponding outputs are the same for all correct processes.

– *Correctness:* 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 for $i = 1, \ldots, k$,

$$(s_i, r_i) \;=\; \mathit{execute}(s_{i-1}, o_i)$$

– *Termination:* If a correct process executes a operation, then the operation eventually generates an output.

The standard implementation of a replicated state machine relies on an atomic broadcast protocol to disseminate the requests to all processes [31, 20]. Every process starts from the same initial state and executes all operations in the order in which they are delivered. If all operations are *deterministic* the states of the correct processes never diverge.

2.3 Leader election

Implementations of asynchronous atomic broadcast need to make some synchrony assumptions or employ randomization [17]. A very weak timing assumption that is also available in many practical implementations is an *eventual leader-detector oracle* [12, 20].

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.

In a model without Byzantine faults, 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 [15]. 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.

Hence we assume that the leader should act according to the application and within some time bounds. If the leader performs wrongly or exceeds the allocated time before reaching this goal, then other processes detect this and report it as a failure to the leader detector by filing a complaint. In an asynchronous system with eventual synchrony as considered here, every process always behaves according to the specification and eventually all remote processes also observe this; if such correct behavior cannot be observed from a process, then the process must be faulty.

Note that our 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.

Definition 3 (Byzantine leader detector). A *Byzantine leader detector* (Ω) is defined with these events:

– *Output event:* Ω-*trust*(p): Indicates that process p is trusted to be leader.
– *Input event:* Ω-*complain*(p): Expresses a complaint about the performance of leader process p.

The primitive satisfies the following properties:

– *Eventual accuracy:* There is a time after which every correct process trusts some correct process.
– *Eventual succession:* If more than f correct processes that trust some process p complain about p, then every correct process eventually trusts a different process than p.
– *Coup resistance:* 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.
– *Eventual agreement:* There is a time after which 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 works at a higher level and outputs not only the identity of a leader but also an increasing *epoch number*. Informally, 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, epoch change is defined as follows [5]:

Definition 4 (Byzantine epoch-change). A *Byzantine epoch-change* (Ψ) primitive is defined with these events:

– *Output event:* Ψ-*start-epoch*(e, p): Indicates that the epoch with number e and leader p starts.
– *Input event:* Ψ-*complain*(e, p): Expresses a complaint about the performance of leader process p.

The primitive satisfies the following properties:

– *Monotonicity:* If a correct process starts an epoch (e, p) and later starts an epoch (e', p'), then $e' > e$.

– *Consistency:* If a correct process starts an epoch (e, p) and another correct process starts an epoch (e', p') with $e = e'$, then $p = p'$.

– *Eventual succession:* Suppose more than f correct processes have started an epoch (e, p) as their last epoch; when these processes all complain about p, then every correct process eventually starts an epoch with a number higher than e.

– *Coup resistance:* When a correct process that has most recently started some epoch (e_1, p_1) starts a new epoch (e_2, p_2), then at least one correct process has complained about leader p_1 in epoch e_1.

– *Eventual leadership:* There is a time after which every correct process has started some epoch and starts no further epoch, such that the last epoch started at every correct process is epoch (e, p) and process p is correct.

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. The value of p_0 is made available to all processes implicitly. 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. [5, Chap. 5].

2.4 Hash functions and digital signatures

We model cryptographic *hash functions* and *digital signature schemes* as ideal, deterministic functionalities implemented by a distributed oracle.

A cryptographic *hash function* maps a bit string of arbitrary length to a short, unique representation. The functionality provides only a single operation *hash*; its invocation takes a bit string x as parameter and returns an integer h with the response. The implementation maintains a list L of all x that have been queried so far. When the invocation contains $x \in L$, then *hash* responds with the index of x in L; otherwise, *hash* appends x to L and returns its index. This ideal implementation models only collision resistance but no other properties of real hash functions.

The functionality of the *digital signature scheme* provides two operations, $sign_p$ and $verify_p$. The invocation of $sign_p$ specifies a process p, takes a bit string m as input, and returns a signature $\sigma \in \{0, 1\}^*$ with the response. Only p may invoke $sign_p$. The operation $verify_p$ takes a putative signature σ and a bit string m as parameters and returns a Boolean value with the response. Its implementation satisfies that $verify_p(\sigma, m)$ returns TRUE for any process p and $m \in \{0, 1\}^*$ if and only if p has executed $sign_p(m)$ and obtained σ before; otherwise, $verify_p(\sigma, m)$ returns FALSE. Every process may invoke *verify*. The signature scheme may be implemented analogously to the hash function.

3 Modular protocol

In this section we discuss the *modular approach* to replicated execution of 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. With the modular approach any application can be replicated without change, and its developers may not 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.

In principle 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 second approach and uses speculative execution.

3.1 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 approach of *Sieve* prevents uncoordinated speculative request execution, which may cause contention among requests from different self-proclaimed leaders and can prevent liveness easily.

The details of Protocol *Sieve* are shown in Algorithms 1–2. The description 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.

3.2 Correctness

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.*

Proof sketch. The *agreement* condition of Definition 2 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

Algorithm 1 Protocol *Sieve*: replicated state machine with non-deterministic operations

```
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 $p_0$
    $next\text{-}epoch$: announced sieve-config number, initially $\perp$     $next\text{-}leader$: announced 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 $rsm\text{-}execute(o)$ do
    $\mathcal{I} \leftarrow \mathcal{I} \cup \{o\}$
    send message $[\text{INVOKE}, config, o]$ over point-to-point link to $leader$

upon receiving message $[\text{INVOKE}, c, o]$ 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 $[\text{EXECUTE}, config, cur]$ over point-to-point links to all processes

upon receiving message $[\text{EXECUTE}, c, o]$ from process $p$ such that $p = leader$ and $c = config$ and $t = \perp$ do
    $(t, r) \leftarrow execute(s, o)$
    $\sigma \leftarrow sign_{self}(\text{SPECULATE} \| config \| hash(t \| r))$
    send message $[\text{APPROVE}, config, o, hash(t \| r), \sigma]$ to $leader$

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

upon event $abv\text{-}deliver(p, [\text{ORDER}, decision, c, o, t_c, r_c, \cdot])$ such that $c = config$ do        // commit operation $o$
    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 $decision = \text{CONFIRM}$ then
        $s \leftarrow t_c$                      // adopt the agreed-on state and response, needed if $(t_c, r_c) \neq (t, r)$
        $rsm\text{-}output(o, s, r_c)$
    $t \leftarrow \perp$

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

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

periodically do
    for every operation $o \in \mathcal{I}$, determine the age of $o$ since it has been invoked and added to $\mathcal{I}$
    if there are “old” operations in $\mathcal{I}$ then
        $\Psi\text{-}complain(leader)$
```

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

```
upon invocation $V(m)$ do
    if $m = [\textsc{order}, \textsc{decision}, c, o, \mathcal{M}]$ then
        if $\mathcal{M}$ is a set of $f+1$ messages of the form $[\textsc{approve}, c_p, o_p, h_p, \sigma_p]$ such that
            $c_p = \mathit{config}$ and $o_p = o$ and $\mathit{verify}_p(\sigma_p, \textsc{speculate}\|c_p\|h_p) = \textsc{True}$ and
            all $h_p$ values in $\mathcal{M}$ are equal then
        return TRUE
    else if $m = [\textsc{order}, \textsc{abort}, c, o, \mathcal{M}]$ then
        if $\mathcal{M}$ is a set of $2f+1$ messages of the form $[\textsc{approve}, c_p, o_p, h_p, \sigma_p]$ such that
            $c_p = \mathit{config}$ and $o_p = o$ and $\mathit{verify}_p(\sigma_p, \textsc{speculate}\|c_p\|h_p) = \textsc{True}$ and
            no $f+1$ of the $h_p$ values in $\mathcal{M}$ are equal then
        return TRUE
    else if $m = [\textsc{new-sieve-config}, c, p]$ then
        if $c \leq \mathit{next\text{-}epoch}$ and $p = \mathit{next\text{-}leader}$ then
            return TRUE
    return FALSE
```

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. □

3.3 Optimizations

Rollback and state transfer. In Protocol *Sieve* every process maintains a copy of the application state resulting from an operation until the operation is committed. Moreover, the confirmed state and the response of an operation are included in ORDER messages that are *abv-broadcast*. For practical applications though, this is often too expensive, and another way for recovering the application state is needed. The solution is to roll back an operation and to transfer the correct state from other processes.

We assume that there exists a *rollback* primitive, ensuring that for $(t, r) \leftarrow \mathit{execute}(s, o)$, the output of $\mathit{rollback}(t, r, o)$ is always s. In the ORDER messages of the protocol, the resulting output state and response are replaced by their hashes for checking the consistency among the processes. Thus, when a process receives an ORDER message with a confirmed operation and hashes ht and hr of the output state and response, respectively, it checks whether the speculative state and response satisfy $\mathit{hash}(t) = ht$ and $\mathit{hash}(r) = hr$. If so, it proceeds as in the protocol.

If the committed operation was aborted, or the values do not match in a confirmed operation, the process rolls back the operation. For a confirmed operation, the process then invokes *state transfer* and retrieves the correct state t and response r that match the hashes from other clients. It will then set the state variable s to t and output the response r. Rollback helps to implement transfer state efficiently, by sending incremental updates only.

For transferring the state, the process sends a STATE-REQUEST message to all those processes who produced the SPECULATE-signatures contained in the $f+1$ APPROVE messages, which the process receives together with a committed and confirmed operation. Since at most f of them may fail to respond, the process is guaranteed to receive the correct state.

State transfer is also initiated when a new configuration starts through an *abv-delivered* NEW-SIEVE-CONFIG message, but the process has already speculatively executed an operation in the last configuration without committing it (this can be recognized by $t \neq \bot$). As in the above use of state transfer, the operation must terminate before the process becomes ready to execute further operations from EXECUTE messages.

Synchronization with PBFT-based atomic broadcast. When the well-known *PBFT protocol* [10] is used to implement *abv-broadcast*, two further optimizations are possible, mainly because PBFT also relies on a leader and already includes Byzantine epoch-change. Hence assume that every process runs PBFT according to Castro and Liskov [10, Sec. 4].

First, let an epoch-change event of Ψ occur at every view change of PBFT. More precisely, whenever a process has received a correct PBFT-NEW-VIEW message for PBFT-view number v and new primary process p, and when the matching VIEW-CHANGE messages have arrived, then the process triggers a Ψ*-start-epoch*(v, p) event at *Sieve*. The process subsequently starts executing requests in the new view. Moreover, complaints from *Sieve* are handled in the same way as when a backup in PBFT *suspects* the current primary to be faulty, namely, it initiates a view change by sending a VIEW-CHANGE message.

The view change mechanism of PBFT ensures all properties expected from Ψ, as follows. The *monotonicity* and *consistency* properties of Byzantine epoch-change follow directly from the calculation of strictly increasing view numbers in PBFT and the deterministic derivation of the PBFT-primary from the view. The *eventual leadership* condition follows from the underlying timing assumption, which essentially means that timeouts are increased until, eventually, every correct process is able to communicate with the leader, the leader is correct, and no further epoch-changes occur.

The second optimization concerns the NEW-SIEVE-CONFIG message. According to *Sieve* it is *abv-broadcast* whenever a new sieve-leader is elected, by that leader. As the leader is directly mapped to PBFT's primary here, it is now the primary who sends this message as a PBFT request. Note that this request might not be delivered when the primary fails, but it will be delivered by the other processes according to the properties of *abv-broadcast*, as required by *Sieve*. Hence, the new sieve-configuration and sieve-leader are assigned either by all correct processes or by none of them.

With these two specializations for PBFT, *Sieve* incurs the additional cost of the EXECUTE/APPROVE messages in the request flow, and one NEW-SIEVE-CONFIG following every view-change of PBFT. But determining the sieve-leader and implementing Ψ do not lead to any additional messages.

3.4 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 may be impossible to change a given application.

Liskov [25] 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 [23] 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.

4 Master-slave protocol

By adopting a *master-slave* approach 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 approach 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 approach 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.

4.1 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:

$$\mathit{nondet\text{-}execute}(s, o) \;\rightarrow\; (s', r, e).$$

Its parameters s, o, s', and r are the same as for *execute*; additionally, the function also outputs *evidence* e. 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

$$\mathit{verify\text{-}execution}(s, o, s', r, e) \;\rightarrow\; \{\textsc{False}, \textsc{True}\}$$

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

4.2 Replication protocol

Implementing a replicated state machine with non-deterministic operations using the master-slave approach 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 $(s', r, e) \leftarrow$ *nondet-execute*(s, o) and *abv-broadcasts* an ORDER message containing the current epoch c and parameters o, s', r, and e. 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, e) = $ 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.

As discussed in the first optimization for *Sieve* (Section 3.3), the output state s' and response r do not always have to be included in the ORDER messages. In the master-slave approach, they can be replaced by hashes only for those operations where the evidence e contains sufficient data for a process to compute the same s' and r values as the master. This holds, for example, when all non-deterministic choices of an operation are contained in e.

Should the master *abv-broadcast* an operation with evidence that does not execute properly, i.e., *verify-execution*$(s, o, s', r, e) = $ FALSE, the atomic broadcast primitive ensures that it is not *abv-delivered* through the external validity property. As in *Sieve*, every process periodically checks if the operations that it has invoked have been executed and complains against the current master using Ψ. This ensures that misbehaving masters are eventually replaced.

4.3 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. This works well in practice for applications in which only a few, known steps can lead to divergence. For example, operations reading inputs from the local system, accessing platform-specific environment data, or generating randomness can be replicated whenever those functions are provided by programming libraries. Master-slave replication may only be employed when the application developer is aware of the causes of non-determinism; for example, a multi-threaded application influenced by a non-deterministic scheduler could not be replicated unless the developer can also control the scheduling (e.g., [22]).

5 Cryptographically secure protocols

Security functions implemented with cryptography are more important today than ever and often found in Internet services. 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. If the service is replicated naively and a key becomes known to all processes, then security effectively decreases.

Thus, 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* [14], 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 approach was pioneered by Reiter and Birman [30] and exploited, for instance, by SINTRA [4, 8] or COCA [36].

In this section we discuss two methods for integrating non-deterministic cryptographic operations in a BFT system. The first scheme is novel in the context of BFT systems and uses verifiable random functions to generate pseudorandom bits that are unpredictable and cannot be biased by a Byzantine process. The second scheme is the well-known approach of using distributed cryptography, as discussed above, which addresses a broad range of cryptographic applications. Both schemes adopt the master-slave replication protocol from the previous section.

5.1 Randomness from verifiable random functions

A *verifiable random function (VRF)* [28] 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.

More precisely, the process owning the VRF chooses a secret seed sk and publishes a public verification key vk. Then the function family $G_{sk} : \{0,1\}^{\lambda} \rightarrow \{0,1\}^{\mu}$ and algorithms P_{sk} and V_{pk} are a VRF whenever these properties hold:

– *Correctness:* $y \leftarrow G_{sk}(x)$ can be computed efficiently from sk and for every x one can also (with the help of sk) efficiently generate a proof $\pi \leftarrow P_{sk}(x)$ such that $V_{pk}(x, y, \pi) = \text{TRUE}$.

– *Uniqueness:* For every input x there is a unique y that satisfies $V_{pk}(x, y, \pi)$, i.e., it is impossible to find y_0 and $y_1 \neq y_0$ and π_0 and π_1 such that $V_{pk}(x, y_0, \pi_0) = V_{pk}(x, y_1, \pi_1) = \text{TRUE}$.

– *Pseudorandomness:* From knowing vk alone and sampling values from V_{sk} and P_{sk}, no polynomial-time adversary can distinguish the output of $G_{sk}(x)$ from a uniformly random μ-bit string, unless the adversary calls the owner to evaluate V_{sk} or P_{sk} on x.

Thus, a VRF generates a value for every input x which is unpredictable and pseudorandom to anyone *not* knowing sk. As $G_{sk}(x)$ is unique for a given x, even an adversarially chosen key preserves the pseudorandomness of G's outputs towards other processes.

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 [26, 21]. In practice, when adopting the random-oracle model, VRFs can immediately be obtained from unique signatures such as ordinary RSA signatures [26].

Replication with cryptographic randomness from a VRF. With the master-slave replication approach, 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.

In more detail, the master computes all random choices while executing an operation o as $r \leftarrow G_{sk}(\mathit{tag})$, where *tag* denotes a unique identifier for the instance and operation. This tag must never reused by the protocol and should not be under the control of the master. The master supplies $\pi \leftarrow P_{sk}(\mathit{tag})$ to the other processes as evidence for the choice of r. During the verification step in *verify-execution*() every process now validates that $V_{pk}(\mathit{tag}, r, \pi) = \text{TRUE}$. When executing the operation, every process uses the same randomness r.

The pseudorandomness property of the VRF ensures that no process apart from the master (or anyone knowing its secret seed) can distinguish r from truly random bits. This depends crucially on the condition that *tag* is used only once as input to the VRF. Hence this approach yields a deterministic pseudorandom output that achieves the desired unpredictability and randomness in many cases, especially against entities that are not part of the BFT system. Note that simply handing over the seed of a cryptographic pseudorandom generator to all processes and treating the generator as part of a deterministic application would be predictable for the slave processes and not pseudorandom.

Of course, if the master is faulty then it can predict the value of r, leak it to other processes, and influence the protocol accordingly. The protocol should leave as little as possible choice to the master for influencing the value of *tag*. It could be derived from an identifier of the protocol or BFT system "instance," perhaps including the identities of all processes, followed by a uniquely determined binary representation of the operation's sequence number. If one assumes all operations are represented by unique bit strings, a hash of the operation itself could also serve as identifier.

5.2 Using distributed cryptography

Distributed cryptography or, more precisely, *threshold cryptography* [14] distributes the power of a cryptosystem among a group of n processes such that it tolerates f faulty ones, which may leak their secrets, fail to participate in protocols, or even act adversarially against the other processes. Threshold cryptosystems extend cryptographic secret sharing schemes, which permit the process group to maintain a secret such that f or fewer of them have no information about it, but any set of *more* than f can reconstruct it.

A *threshold public-key cryptosystem (T-PKCS)*, for example, is a public-key cryptosystem with distributed control over the decryption operation. There is a single public key for encryption, but each process holds a *key share* for decryption. When a ciphertext is to be decrypted, every process computes a decryption share from the ciphertext and its key share. From any $f + 1$ of these decryption shares, the plaintext can be recovered. Usually the decryption shares are accompanied by zero-knowledge proofs to make the scheme robust. This models a non-interactive T-PKCS, which is practical because it only needs one round of point-to-point messages for exchanging the decryption shares; other T-PKCSs require interaction among the processes for computing shares. The public key and the key shares are generated either by a trusted entity before the protocol starts or again in a distributed way, tolerating faulty processes that may try to disrupt the key-generation protocol [18].

A state-of-the-art T-PKCS is secure against *adaptive chosen-ciphertext attacks* [34], ensuring that an adversary cannot obtain any meaningful information from a ciphertext unless at least one correct process has computed a decryption share. With a T-PKCS the BFT system can receive operations in encrypted form, order them first without knowing their content, and only decrypt and execute them after they have been ordered. This approach defends against violations of the causal order among operations [30, 6].

A *threshold signature scheme* works analogously and can be used, for instance, to implement a secure name service or a certification authority [4, 36, 9]. Practical non-interactive threshold signature schemes are well-known [33]. To generate cryptographically strong and unpredictable pseudorandom bits, *threshold coin-tossing schemes* have also been constructed [7]. They do not suffer from the limitation of the VRF construction in the previous section and ensure that no single process can predict the randomness until at least one correct process has agreed to start the protocol.

Replication with threshold cryptosystems. Threshold cryptosystems have been used in BFT replication starting with the work Reiter and Birman [30]. Subsequently SINTRA [8] and other systems exploited it as well with robust, asynchronous protocols.

For integrating a threshold cryptosystems with a BFT system, no particular assumptions are needed about the structure of the atomic broadcast or even the existence of a leader. The distributed scheme can simply be inserted into the code that executes operations and directly replaces the calls to the cryptosystems.

To be more precise, suppose that *execute*(s, o) invokes a call to one of the three cryptosystem functions discussed before (that is, public-key decryption, issuing a digital signature, or generating random bits). The process now invokes the threshold algorithm to generate a corresponding share. If the threshold cryptosystem is non-interactive, the process sends this share to all others over the point-to-point links. Then the process waits for receiving $f + 1$ shares and assembles them to the result of the cryptographic operation. With interactive threshold schemes, the processes invoke the corresponding protocol as a subroutine. Ideally the threshold cryptosystem supports the same cryptographic signature structure or ciphertext format as the standardized schemes; then the rest of the service (i.e., code of the clients) can remain the same as with a centralized service. This holds for RSA signatures, for instance [9].

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*, 2015, pp. 104–121.

[2] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault-tolerance," *ACM Transactions on Computer Systems*, vol. 14, no. 1, pp. 80–107, Feb. 1996.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," in *Distributed Systems (2nd Ed.)*, S. Mullender, Ed. New York: ACM Press & Addison-Wesley, 1993.

[4] C. Cachin, "Distributing trust on the Internet," in *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, 2001, pp. 183–192.

[5] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.

[6] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols (extended abstract)," in *Advances in Cryptology: CRYPTO 2001*, ser. Lecture Notes in Computer Science, J. Kilian, Ed., vol. 2139. Springer, 2001, pp. 524–541.

[7] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography," *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[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)*, Jun. 2002, pp. 167–176.

[9] C. Cachin and A. Samar, "Secure distributed DNS," in *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, Jun. 2004, pp. 423–432.

[10] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[11] M. Castro, R. Rodrigues, and B. Liskov, "BASE: Using abstraction to improve fault tolerance," *ACM Transactions on Computer Systems*, vol. 21, no. 3, pp. 236–269, 2003.

[12] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[13] 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.

[14] Y. Desmedt, "Threshold cryptography," *European Transactions on Telecommunications*, vol. 5, no. 4, pp. 449–457, 1994.

[15] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper, "Muteness failure detectors: Specification and implementation," in *Proc. 3rd European Dependable Computing Conference (EDCC-3)*, ser. Lecture Notes in Computer Science, J. Hlavicka, E. Maehle, and A. Pataricza, Eds., vol. 1667. Springer, 1999, pp. 71–87.

[16] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[18] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," *Journal of Cryptology*, vol. 20, pp. 51–83, 2007.

[19] 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.

[20] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems*, S. J. Mullender, Ed. New York: ACM Press & Addison-Wesley, 1993, expanded version appears as Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.

[21] T. Jager, "Verifiable random functions from weaker assumptions," in *Proc. 12th Theory of Cryptography Conference (TCC 2015)*, ser. Lecture Notes in Computer Science, Y. Dodis and J. B. Nielsen, Eds., vol. 9015. Springer, 2015, pp. 121–143.

[22] 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.

[23] 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.

[24] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, Jun. 2004, pp. 575–584.

[25] B. Liskov, "From viewstamped replication to Byzantine fault tolerance," in *Replication: Theory and Practice*, ser. Lecture Notes in Computer Science, B. Charron-Bost, F. Pedone, and A. Schiper, Eds. Springer, 2010, vol. 5959, pp. 121–149.

[26] A. Lysyanskaya, "Unique signatures and verifiable random functions from the DH-DDH separation," in *Advances in Cryptology: CRYPTO 2002*, ser. Lecture Notes in Computer Science, M. Yung, Ed., vol. 2442. Springer, 2002, pp. 597–612.

[27] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press, 1997.

[28] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999, pp. 120–130.

[29] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.

[30] M. K. Reiter and K. P. Birman, "How to securely replicate services," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 986–1009, May 1994.

[31] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[32] U. C. Services, "Allen clement and manos kapritsos and sangmin lee and yang wang and lorenzo alvisi and mike dahlin and taylor riche," in *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 277–290.

[33] V. Shoup, “Practical threshold signatures,” in *Advances in Cryptology: EUROCRYPT 2000*, ser. Lecture Notes in Computer Science, B. Preneel, Ed., vol. 1087. Springer, 2000, pp. 207–220.

[34] V. Shoup and R. Gennaro, “Securing threshold cryptosystems against chosen ciphertext attack,” in *Advances in Cryptology: EUROCRYPT ’98*, ser. Lecture Notes in Computer Science, K. Nyberg, Ed., vol. 1403. Springer, 1998, pp. 1–16.

[35] M. Vukolić, “The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication,” in *Proc. IFIP WG 11.4 Workshop on Open Research Problems in Network Security (iNetSec 2015)*, ser. Lecture Notes in Computer Science. Springer, 2016.

[36] L. Zhou, F. B. Schneider, and R. van Renesse, “COCA: A secure distributed online certification authority,” *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 329–368, 2002.