Byzantine Reliable Broadcast with One Trusted Monotonic Counter

Yackolley Amoussou Guenou[1], Lionel Beltrando[2], Maurice Herlihy[3], and Maria Potop-Butucaru[2]

[1] Université Paris-Panthéon-Assas, CRED, Paris, France
[2] France Sorbonne University, LIP6, Paris, France
[3] Brown University, USA

Abstract. Byzantine Reliable Broadcast is one of the most popular communication primitives in distributed systems. Byzantine reliable broadcast ensures that processes agree to deliver a message from an initiator even if some processes (perhaps including the initiator) are Byzantine. In asynchronous settings it is known since the prominent work of Bracha [4] that Byzantine reliable broadcast can be implemented deterministically if $n \geq 3t + 1$ where t is an upper bound on the number of Byzantine processes. Here, we study Byzantine Reliable Broadcast when processes are equipped with *trusted execution environments* (TEEs), special software or hardware designed to prevent equivocation. Our contribution is twofold. First, we show that, despite common belief, when each process is equipped with a TEE, Bracha's algorithm still needs $n \geq 3t + 1$. Second, we present a novel algorithm that uses a single TEE (at the initiator) that implements Byzantine Reliable Asynchronous Broadcast with $n \geq 2t + 1$.

1 Introduction

Byzantine reliable broadcast is a fundamental problem in fault-tolerant distributed systems. It consists of ensuring that a correct initiator process broadcasts its value to all correct processes, even in the presence of malicious Byzantine processes. Byzantine Reliable Broadcast was for decades at the core of various consensus protocols and more recently at the core of certain blockchains.

Byzantine Reliable Broadcast have been addressed in various settings: with fixed and mobile Byzantine nodes, dynamicity or in conjunction with transient faults. Byzantine Reliable Broadcast solutions (e.g. [2,3,11,13,14]) achieve resilience of at least $n \geq 3t + 1$ processes, where t is the maximum number of Byzantine processes. However, these solutions require strong network assumptions, such as *synchrony* (processes execute in lock-step) or *non-equivocation* (the initiator must send the same message to all processes). More recently, *trusted execution environments* (TEEs) have emerged as a promising protection against Byzantine failures by providing cryptographic primitives that protect participants against equivocation.

TEEs are especially promising for consensus protocols such as PBFT, and therefore for many recent blockchain algorithms. For example, Correia *et al.*[7,8] recently introduced the *TTCB wormhole*, which supports a PBFT protocol that tolerates up to half of the processes to be Byzantine, well beyond the tolerance of classical systems [10]. Nevertheless, the trusted part of this systems makes practical implementations difficult.

A2M (Attested Append-only Memory) [5] provides a small and easy-to-implement abstraction of a trusted append-only log. Each log has a unique identifier, and offers methods to append and read values. A value, once added, cannot be re-written. A2M increases the resilience of PBFT by appending each message to the log and sending that attestation along with the message, which increases resiliency to one-half.

We are not the first to suggest that trusted environments similar to A2M can increase resilience for blockchains: see *HotStuff* [17], *Damysus* [9], and TenderTee [1]. However, none of these works focuses on the Byzantine reliable broadcast primitive. In [1] the authors conjecture that plugging A2M hardware into Bracha's protocol might increase its resilience. In this paper, we refute their conjecture by showing that A2M-Bracha has the same resilience as the original in asynchronous settings.

An alternative to A2M is the use of a *monotonic counter* implemented in a tamper-proof module. TrInc [12] is a trusted component that deals with equivocation in large systems by providing a set of monotonic counters, supported by a trusted hardware unit called a *trinket*.

More recently, Veronese *et al.* [16] propose *USIG* (Unique Sequential Identifier Generator), a service available to each process (and implemented in a tamper-proof module) that assigns each message a unique counter value, and signs that message. The service offers two functions: one that returns a certificate, and one that validates certificates. These certificates are based on a secure counter: the counter value is never duplicated, counter values are increasing, and successive counter values are successive integers. To the best of our knowledge, this kind of TEE has never been used to increase the resilience of reliable broadcast. Here we prove that such a TEE can implement Byzantine Reliable Broadcast in asynchronous environments with optimal resilience.

Our contribution. This paper presents a study of Byzantine Reliable Broadcast using TEEs. First we show that, despite popular belief, TEEs cannot improve the resilience of Bracha's algorithm. Instead, we propose a novel algorithm that uses a single (optimal number) TEE to implement asynchronous Byzantine reliable broadcast with n processes, $n \geq 2t + 1$ where t is an upper bound on the number of Byzantine processes. Interestingly, this algorithm uses only one simple TEE that provides a trusted monotonic counter. We abstract the TEE via a distributed object called Trusted Monotonic Counter Object.

Paper organisation. The paper is organized as follows. Section 2 defines the execution model and presents the specification of the Byzantine Reliable Broadcast problem. Section 3 introduces the key component of our Byzantine Reliable

Broadcast implementation, the Trusted Monotonic Counter Object. Section 4 discusses the impossibility to improve Bracha's Byzantine Reliable Broadcast resilience even when each process is equipped with a TEE. Section 5 presents our algorithm for Byzantine Reliable Broadcast using a single Trusted Monotonic Counter Object at the initiator.

2 System model and Problems Definition

We consider a set of n asynchronous sequential processes. Up to t processes can be *Byzantine*, meaning they can deviate from the given protocol. The rest are *correct* processes.

Processes communicate by exchanging messages through an asynchronous network. We make the usual assumptions that there is a public key infrastructure (PKI) where public keys are distributed, each process has a (universally known) public key, a matching private key, and each message is signed by its creator. Messages are not lost or spuriously generated. Each process can send messages directly to any other process, and each process can identify the sender of every message it receives.

We assume processes have access to a broadcast primitive broadcast(m) which ensures that message m is received by every correct process in a finite (but unknown) time.

Following Bracha, [4], we define *Byzantine Reliable Broadcast* as follows:

Definition 1 (Byzantine Reliable Broadcast problem).

- *brb-CorrectInit: If the initiator is correct, all correct processes deliver the initiator's value.*
- *brb-ByzantineInit: If the initiator is Byzantine, then either no correct process delivers any value, or all correct processes deliver the same value.*

3 Trusted Monotonic Counter Object

TEEs in general (e.g. A2M [5], TrInc [12], USIG [16]) are reputed to be powerful tools for avoiding equivocation. Although the TEE abstraction makes protocols immune to equivocation (where the initiator sends different messages to to different processes), Clement *et al.* [6] show that non-equivocation is not enough to provide $n \geq 2f + 1$ resilience nor to support the equivalent of digital signatures.

We now define the Trusted Monotonic Counter Oracle abstraction TMC-Object the core of our novel Byzantine Reliable Broadcast protocol that supports t Byzantine failures among n processes, where $n \geq 2t + 1$, an improvement on the classical $n \geq 3t + 1$ algorithms.

In short, TMC-Object provides a non-falsifiable, verifiable, unique, monotonic, and sequential counter. In particular, TMC-Object provides each process with a read-only local variable, called trustedCounter. Whenever the TMC-Object is invoked, it returns a value for trustedCounter strictly greater than any previous

value it returned. The difference between two successive counter values is exactly 1, so when a process p receives a two messages stamped with counter values, it can detect whether there have been intermediate messages.

The TMC-Object supports the operation **get_certificate()**. A process p invokes **get_certificate(m)** with a message m. The object returns a *certificate* and a *unique identifier*. The certificate certifies that the returned unique identifier was created by the tamper-proof TMC-Object object for the message m. The unique identifier is essentially a reading of the monotonic counter trustedCounter, incremented whenever **get_certificate(m)** is called.

The TMC-Object object guarantees the following properties:

- **Uniqueness:** TMC-Object will never assign the same identifier to two different messages.
- **Strict Monotonicity:** TMC-Object will always assign an identifier that is strictly greater than the previous one.
- **Sequentiality:** TMC-Object will always assign an identifier that is the successor of the previous one.

4 Bracha's Byzantine Reliable Broadcast with TEEs

In this section we prove that modifying Bracha's reliable broadcast algorithm [4] by (only) equipping each processor with a TEE (here, TMC-Object) does not change the tolerance threshold of Byzantine processes, which remains 1/3.

To send a message u certified with TMC-Object, a process p first invokes TMC-Object, which creates a certificate $\mathcal{C}_p$ corresponding to the value of the trustedCounter c_p, then the process sends the tuple $(u, \mathcal{C}_p, c_p)$, which can be verified by any other process receiving the message. Each invocation to TMC-Object increments the value of the trustedCounter c_p of process p. In the following, that sequence of operation is simply called **TMC-Object-Send** u.

In the following, we describe Algorithm 1 which is Bracha's reliable broadcast where each process uses the TMC-Object-Send operation instead of a Send operation. In more detail, the protocol works in sequential steps. In the broadcast primitive described in Algorithm 1, there are three types of messages used in the protocol: *initial*, *echo*, and *ready*. All these messages are sent using the TMC-Object-Send operation. In the initial step (Step 0) of the protocol, when a process p wants to broadcast a value u, it TMC-Object-Sends an initial message for u ($< initial, u >$) to all other processes. The process initiating the broadcast is called the *initiator*.

In Step 1, when receiving a valid[4] initial message with value v from the initiator, a process A2M-Sends an echo message for v ($< echo, v >$). An echo message is also sent if instead of receiving the initial message, the process receives

[4] Here, valid TMC-Object-message, or TMC-Object-first message means that the message is the first TMC-Object-Send operation done by the initiator. If the value of the counter is strictly greater than 2, the initiator may have equivocated (having already A2M-sent another message).

enough (here, α) echo messages for the same value from different processes, implying that many processes saw the initiator message. After, and only after the A2M-Send operation, the process moves to Step 2.

In Step 2, each process waits to receive echo messages for the same value, say v, from at least α different processes sent from Step 1. When that is the case, the process TMC-Object-Sends a ready message for the value v ($< ready, v >$). After the send operation, the process moves to Step 3.

Step 3 is similar to Step 2. The process waits for β ready messages, when the β ready messages are received for the same value, say v, the process delivers value v and finishes the instance of broadcast.

The broadcast is successful if all the correct processes rb-Deliver the same value. Thus, rb-Broadcast and rb-Deliver provide us a pair of communication primitives.

Algorithm 1 Reliable Broadcast with a Trusted Environment

```
1: procedure RB-BROADCAST(u)
2:     Step 0
3:       if p is the initiator then
4:         TMC-Object-Send < initial, u > to all
5:     Step 1
6:       Wait until receipt of
7:         1 TMC-Object-first < initial, v > message, or
8:         α < echo, v > messages
9:         for some v
10:      TMC-Object-Send < echo, v > to all
11:    Step 2
12:      Wait until receipt of
13:        α < echo, v > messages
14:          (including messages received in Step 1)
15:        for some v
16:      TMC-Object-Send < ready, v > to all
17:    Step 3
18:      Wait until receipt of
19:        β < ready, v > messages
20:          (including messages received in Steps 1 and 2)
21:        for some v
22:      rb-Deliver v
```

Lemma 1. *Consider Algorithm 1 with parameter $n \geq \alpha > t$ and $\alpha \geq n/2 + 1$ where t is the number of Byzantine processes. If two correct processes TMC-Object-Send $< echo, v >$ and $< echo, u >$ messages, respectively, then $u = v$.*

Proof. The proof will be conducted by contradiction. Assume there exist two correct processes that TMC-Object-Send $< echo, v >$ and $< echo, u >$ messages, respectively, with $u \neq v$. Let q be the first correct process that TMC-Object-Sends

an $< echo, v >$ message, and let r be the first correct process that TMC-Object-Sends an $< echo, u >$ message.

- Case 1: Process q receives an initial message $< initial, v >$ and process r receives an initial message $< initial, u >$. If the initiator is correct, then this situation is impossible since a correct initiator sends only one initial value. If the initiator is Byzantine, then either q or r rejects the initial value since the TMC-Object nominal sequence is invalid (the counter associated with one of these values is strictly greater than 1, hence the message is not valid).
- Case 2: Process q must have received $\alpha < echo, v >$ messages, and process r must have received $\alpha < echo, u >$ messages. Notice that a correct process can send only one echo. Since $\alpha > t$, where t is the number of Byzantine processes in the system, among the α messages some come from correct processes. Since $\alpha \geq n/2 + 1$ then there is at least one correct process that TMC-Object-Sent $< echo, v >$ and $< echo, u >$ messages which is impossible, since they are correct.

□

Lemma 2. *Consider Algorithm 1 with parameter $n \geq \alpha > t$ and $\alpha \geq n/2 + 1$ where t is the number of Byzantine processes. If two correct processes TMC-Object-Send $< ready, v >$ and $< ready, u >$ messages, respectively, then $u = v$.*

Proof. Proof by contradiction. Assume there exist two correct processes which TMC-Object-Send $< ready, v >$ and $< ready, u >$ messages, with $u \neq v$. Let q be the first process that TMC-Object-Sends a $< ready, v >$ message, and let r be the first process that TMC-Object-Sends a $< ready, u >$ message. Process q must have received more than $\alpha < echo, v >$ messages, and process r must have received more than $\alpha < echo, u >$ messages. Since $\alpha > t$ and $\alpha \geq n/2 + 1$ it follows that some correct process must have TMC-Object-Sent $< echo, v >$ and some correct process must have TMC-Object-Sent $< echo, u >$ messages. Following Lemma 1, we then have $u = v$. □

Lemma 3. *Consider Algorithm 1 with parameter $n \geq \alpha > t$ and $\alpha \geq n/2 + 1$ where t is the number of Byzantine processes. If two correct processes, q and r, deliver the values v and u, respectively, then $u = v$.*

Proof. If q delivers the value v then it must have received $\alpha < ready, v >$ messages, and therefore a $< ready, v >$ messages is from at least 1 a correct process. Similarly, r must have received a $< ready, u >$ messages from at least 1 correct process. By Lemma 2, $u = v$. □

We now show that Algorithm 1 satisfies property brb-CorrectInit of the Byzantine reliable broadcast.

Theorem 1. *Consider Algorithm 1 with parameters α and β. If the initiator is a correct process Algorithm 1 satisfies the brb-CorrectInit property if $\alpha = \beta$ and $n/2 + 1 \leq \alpha \leq n$ and $n \geq 2t + 1$ where t is the number of Byzantine processes and n the total number of processes.*

Proof. The proof follows directly from Lemmas 1, 2 and 3. Let p be the initiator. Since the initiator is correct, the value broadcast by p, u, will eventually be received by all other correct processes (at least $n - t = t + 1$ correct processes). These processes will echo that value u (TMC-Object-Send an echo message for u). Since all correct processes echo the same value (Lemma 1), and their number is sufficient to make the protocol advance (there are at least $n/2+1$ correct processes), each correct process will receive enough echoes to send a ready message, and the same one (Lemma 2). By the same argument and applying Lemma 3, all correct processes will receive enough ready messages for the initiator value u, and then will rb-Deliver the initiator message. □

Unfortunately, the following result shows that in the presence of a Byzantine initiator, Algorithm 1 could produce undesirable behaviour, hence does not implement the Byzantine reliable broadcast.

Lemma 4. *Let n be the number of processes, and t be an upper bound on the Byzantine processes with $n \geq 2t + 1$. Consider Algorithm 1 with parameters α and β with $\alpha = t + 1$ and $t + 1 \leq \beta < 2t + 1$ Algorithm 1 does not satisfy the brb-ByzantineInit property when the initiator is Byzantine.*

Proof. If the initiator is a Byzantine process, Byzantine processes could force a subset of correct processes to deliver a value, and another subset of correct processes to never deliver any value. Note that even though all processes use a TMC-Object abstraction such that Byzantine processes cannot equivocate, Byzantine processes still can send a message to some processes but not to others.

Let p be the Byzantine initiator. p TMC-Object-sends a value u to $1 \leq x \leq t$ correct processes $q_1, q_2 \dots q_x$ but not to the other $n - t - x$ processes. Denote by Q this set of x correct processes receiving the initiator's initial message. Since processes in Q receive the message from the Byzantine initiator, they TMC-Object-Send an echo message for u. Assume now that all the Byzantine processes TMC-Object-Send an echo message for value u only to processes in Q but not to the others. It follows that all correct processes but those in Q have no message from the initiator and only the $< echo, u >$ from processes in Q. Those processes cannot advance past Line 6 of Algorithm 1 since they need at least $\alpha = t + 1 > x$ echo messages.

All correct processes but those in Q have only x echo messages that come from the processes in Q. Processes in Q on the other hand would have the echo messages from all Byzantine processes in addition to their own echo messages, which sums to $t+x$ echo messages for the value u. Therefore, processes in Q will advance and TMC-Object-Send a ready message for u.

In the same spirit, Byzantine processes can TMC-Object-Send a ready message for value u to processes in Q only. The other correct processes are still blocked at 6 of Algorithm 1. In addition to the ready messages from the Byzantine processes, processes in Q also get their own ready messages for value u, so each processes $q \in Q$ has a total of $t + x$ ready messages and hence delivers the value u (rb-Delivery). The other correct processes only receive the values TMC-Object-Sent by processes in Q, meaning x echo message and x ready message,

both for u, hence, they can never reach a acceptance decision in Algorithm 1. It follows that Algorithm 1 does not satisfy the brb-ByzantineInit property when the initiator is Byzantine. □

When $\alpha = \beta = t + 1$, Lemma 4 violates the brb-ByzantineInit property of the Byzantine reliable broadcast (Definition 1), hence, does not satisfy the Byzantine reliable broadcast (Definition 1) as stated by the following Corollary.

Corollary 1. *Let n be the number of processes, and t an upper bound of the Byzantine processes. If $n \geq 2t+1$, Algorithm 1 does not implement the Byzantine reliable broadcast.*

However, for Algorithm 1 to implement the Byzantine reliable broadcast, we show in Theorem 2 that we must have $\beta = 2t + 1$.

Theorem 2. *Necessary conditions for 1 with parameters α and β to implement the Byzantine reliable broadcast are: $\alpha = t+1$, $\beta = 2t+1$, and $n-t \geq \beta$, where t is the upper bound of the number of Byzantine processes and n the total number of processes.*

Proof. If the initiator is correct, all correct processes decide to deliver the initiator message, by Theorem 1.

It remains to show that when the initiator is Byzantine, either no correct process delivers any value or all correct processes deliver the same value.

- By Lemma 3, if two correct processes deliver a value, they must deliver the same one.
- Now, let us turn to the case where only one correct process reaches a decision. Assume that process q reaches a decision and delivers a value u. It means that q received at least β ready messages, from which at least $\beta - t$ are from correct processes.
  At least $\beta - t$ correct processes sent a ready message. However, to TMC-Object-Send a ready message, a correct processes must have reached Step 2, and must have completed Step 1 of Algorithm 1. In fact, if a correct process does not TMC-Object-Send an echo message, it cannot enter Step 2. Therefore, we know that at least $\beta - t$ correct processes have TMC-Object-Sent an echo message. By Lemma 1, all correct processes that TMC-Object-Sent an echo message have TMC-Object-Sent it for the same value, hence it means that they all TMC-Object-Sent an echo message for the value u.
  We would like to have that, with at least $\beta - t$ correct processes TMC-Object-Sending an echo message for the same value, say a value u, all correct processes must have received at least α echo messages for value u. Hence, we have that $\beta - t \geq \alpha \implies \beta \geq \alpha + t \geq 2t + 1$. For lower bounds, now assume that $\alpha = t + 1$ and $\beta = 2t + 1$. The rest of the proof shows that it is sufficient for Algorithm 1 to implement the Byzantine reliable broadcast.
  Hence they will all TMC-Object-Send an echo message for u. In that case, all correct processes (at least $2t + 1$ processes) TMC-Object-Sent an echo for u, all correct processes (at least $2t + 1$) will then TMC-Object-Send a ready

message for u, which leads to all correct processes eventually delivering value u. Hence, if one correct process delivers a value u, all other correct processes eventually deliver the same value u.

□

5 Byzantine Reliable Broadcast with optimal TMC-Object

Algorithm 2 Byzantine Reliable Broadcast with a unique Trusted Environment for the initiator

```
1: procedure BRB-BROADCAST(u)
2:     Step 0
3:       if p is the initiator then
4:         TMC-Object-Send < initial, u, id_initiator > to all
5:     Step 1
6:       Wait until receipt of
7:         1 TMC-Object-first < initial, v, id_initiator > message
8:         for some v
9:       Send < echo_in, v, (< initial, v, id_initiator >, C_initiator, c_initiator) > to all //the process broadcasts back the initiator's message with the associated certificate and trusted counter.
10:    Step 2
11:      Wait until receipt of
12:        t + 1 < echo_in, v > messages    //Projection of the echoes received keeping only the value of the message.
13:          (including messages received in Step 1)
14:        for some v
15:      Send < ready, v > to all
16:    Step 3
17:      Wait until receipt of
18:        t + 1 < ready, v > messages
19:          (including messages received in Steps 1 and 2)
20:        for some v
21:      brb-Deliver v
```

In this section, we present Algorithm 2, which contains a small modification of Bracha's algorithm [4], solving the reliable broadcast problem tolerating $t < n/2$ Byzantine processes. Algorithm 2 uses the Trusted Execution Environment (TEE) setups to increase the security threshold of Byzantine reliable broadcast from 1/3 of Byzantine processes to 1/2.

Moreover, to reduce the use of the TEE, which can be resource-intensive, only the initiator is required to send certified messages. The other processes simply check the validity of the message and its certification, but do not require the equipment to send certified messages. In this section, we consider the use of TMC-Object defined in Section 3.

In Algorithm 2, only the initiator sends a message using the light TMC-Object abstraction for its send operation. We say that the initiator TMC-Object-Sends a message. The other processes send their message classically. The other difference with Algorithm 1 is that during Step 1, when a process receives the initiator message, say for value u, it sends an echo message for u coupled with the initiator message (meaning the message, and the certificate and counter sent by the initiator). In such a way, it ensures that all other processes will eventually receives the initiator certified message. The rest of the algorithm proceed as in Bracha (or in Algorithm 1 where the TMC-Object-Sends are replaced by classical Send operations).

The broadcast is successful if all the correct processes brb-Deliver the same value, say u. Thus, brb-Broadcast and brb-Deliver provide a pair of communication primitives resilient to $t < n/2$ Byzantine processes.

We can now prove the correctness of Algorithm 2 against the Byzantine reliable broadcast abstraction.

Recall that we say that a process *accept*s, a message, if it receives and adds the "valid" messages in term of validity of the TMC-Object, meaning that there were no message before in that same category, hence the value of the trusted counter is the lowest. Notice that if the message received is not expected to be an TMC-Object message, and is indeed not part of a TMC-Object operation (e.g., a send which is not TMC-Object-Send), such a message is valid by default and, therefore, is accepted.

Finally, for any value v, the message $< echo\_in, v, (< initial, v, id\_initiator >, \mathcal{C}_{\text{initiator}}, c_{\text{initiator}}) >$ should be understood as two messages bundled together, i.e., the echo message $< echo, v >$ sent after the reception of the initiator message and sending back the initiator's message $< initial, v, id\_initiator >$, along with the certificates and trusted counter $\mathcal{C}_{\text{initiator}}, c_{\text{initiator}}) >$. The message is exactly the initiator's message, the TMC-Object message will be correctly validated.

Lemma 5. *In any execution of Algorithm 2, with $n \geq 2t + 1$ where t is the number of Byzantine processes, a correct process sends each type of message (initial, echo_in, ready) at most once.*

Proof. In Steps 1 and 2 of Algorithm 2, the protocol requires sending exactly 1 message, then to move to the subsequent phase. A correct process cannot send more messages.

In Step 0, a correct process TMC-Object-Sends a message if and only if it is the initiator, and after that moves to Step 1. If the process is not the initiator, it does not (TMC-Object-)Send anything, but moves directly to Step 1. Hence, in Step 0, at most 1 message is sent. □

Lemma 6. *Consider Algorithm 2 with $n \geq 2t + 1$ where t is the number of Byzantine processes. If two correct processes p and q receive and accept respectively $< initial, u, id\_initiator >$ and $< initial, v, id\_initiator >$, then $u = v$.*

Proof. This holds thanks to the properties of the TMC-Object. Since equivocation is not possible at the initiator level, thanks to the use of the counter in TMC-Object, if two correct processes p and q accept an initiator message, then they

received the same message, and they then echo the accepted initial message (Line 9 of Algorithm 2). □

By Lemma 6, we know that if two correct processes accept an initiator message, then they accept the same message. The only thing that could happen is for one correct process to receive the initiator message, while another process does not receive such a message.

Lemma 7. *Consider Algorithm 2 with $n \geq 2t + 1$ where t is the number of Byzantine processes. If one correct process sends $< echo\_in, v >$ for some v, then all correct process will eventually send $< echo\_in, v >$.*

Proof. Let p and q be processes. Without loss of generality, assume that p is the first correct process to do an echo. If a correct process echo a message, it means it accepted the initiator message, and will send the echo along with the TMC-Object-Send of the initiator (Line 9 of Algorithm 2). Two cases can arise. Either the initiator sent the initial message to both p and q, or the initiator did not send a message to q. Notice that it is not possible for the initiator not to have TMC-Object-Sent a message to p, since p echoed the initiator message.

First, consider the case where p received the initiator message, but not q. The process p sent an echo message containing the initiator's initial message respecting the TMC-Object format. By assumption, a message sent by a correct process will eventually be received by all the other correct processes. Therefore, eventually q will receive p's echo message, containing the initial message. q will be able to assess the validity of the initial message (according to the initiator signatures), will accept it, and will send an echo for the message too.

Finally, if the initiator sent a message to both p and q, therefore, either it sends the same message to p and q, and so q will echo that same message (by Lemmas 5 and 6, it is not possible for q to echo something else), or the message sent to q is invalid and not accepted. That last case is equivalent to the above situation, since an invalid message is not registered nor considered, and is equivalent to not have received a message. □

Thanks to Lemmas 5 and 7, we know that whenever a correct process sends an *echo_in* message, all other correct processes will also echo a message (and more accurately, the same message).

Lemma 8. *If two correct processes p and q send respectively $< ready, v >$ and $< ready, u >$, then $u = v$.*

Proof. By contradiction. Assume $u \neq v$ are two messages. Without loss of generality, let p be the a process that sends a $< ready, v >$ message, and let q be a process that sends a $< ready, u >$ message. To send a ready message for value x, a correct process must have received from at least $t + 1$ different processes the message (echo_in, x) (Line 15 of Algorithm 2). Therefore, p must have received the message $< echo\_in, v >$ from at least $t + 1$ different processes, and process q must have received the message $< echo\_in, u >$ from at least $t + 1$ different processes. Since there are at most t Byzantine processes, at least one correct

process must have sent an echo message for both u and v, which is impossible by Lemma 5. Therefore, it is impossible to have $u \neq v$. □

Lemma 9. *Consider Algorithm 2 with $n \geq 2t + 1$ where t is the number of Byzantine processes. If two correct processes, p and q, brb-deliver the values v and u, respectively, then $u = v$.*

Proof. This proof is similar to the proof for 8. We proceed by contradiction. Assume two messages u and v such that $u \neq v$. Without loss of generality, let p be the a process that brb-delivers v, and let q be a process that brb-delivers u. To brb-deliver a value x, a correct process must have received from at least $t+1$ different processes the message $< \mathit{ready}, x >$ (Line 18 of Algorithm 2). Therefore, p must have received the message $< \mathit{ready}, v >$ from at least $t + 1$ different processes, and process q must have received the message $< \mathit{ready}, u >$ from at least $t + 1$ different processes. Since there are at most t Byzantine processes, it means that at least one correct process sent a ready message for both u and v, which is impossible by Lemma 5. Therefore, it is impossible to have $u \neq v$. □

Lemma 10. *Consider Algorithm 2 with $n \geq 2t + 1$ where t is the number of Byzantine processes. If a correct process p delivers the value v then every other correct process will eventually deliver v.*

Proof. If p brb-Deliver v then p received the message $< \mathit{ready}, v >$ from at least $t+1$ different processes. Since there are at most t Byzantine processes, it means that at least one correct process sent a message $< \mathit{ready}, v >$. Since one correct process sent a ready message for v, it means that it received an $< \mathit{echo}, v >$ message from at least $t+1$ different processes; hence, (since there are at most t Byzantine processes) it means that at least one correct process sent a message $< \mathit{echo}, v >$. Therefore, by Lemma 7, all other correct processes will (eventually) send an echo message for v. Those will be received by all the correct processes. This will lead to having at least $t + 1$ different processes sending it. All correct processes will, therefore, eventually send a ready message for v (by Lemma 8, since we already know that one correct sent a ready for v). Hence, at least $t+1$ ready messages will be received by all correct processes, that will lead them to brb-Deliver v. □

Lemma 11. *Consider Algorithm 2 with $n \geq 2t + 1$ where t is the number of Byzantine processes. If a correct process p broadcasts v then all correct processes brb-deliver v.*

Proof. The proof of the lemma is straightforward. If a correct process broadcasts a initial message, it does so to all processes. All processes in Step 1 will echo_in the initiator message v, thanks to Lemma 7. Since correct processes are the majority, and network is eventually synchronous, they will all eventually receive at least $t+1$ echo_in message for v and send each a ready message for v. Thanks to Lemma 8, since one correct process sends a ready for v, v is the only value correct process will send a ready for. That value will, therefore, be present in at least $t + 1$ ready messages, hence in Step 3 a correct process will brb-deliver v. By Lemma 10, all correct processes will eventually brb-deliver v.

We can now prove that the algorithm implements Byzantine reliable broadcast.

Theorem 3. *Let n be the number of processes, and t an upper bound of the Byzantine processes. If $n \geq 2t + 1$, Algorithm 2 implements Byzantine reliable broadcast.*

Proof. By Lemma 11, when a correct initiator broadcasts a value, all correct processes brb-deliver that value.

By Lemma 7, if a correct process brb-delivers an initiator message (even if the initiator is Byzantine), all correct process will eventually brb-deliver the same initiator message. In that case, the rest of the proof follows thanks to Lemma 11.

Otherwise, no correct process brb-delivers any value. In more details, if a Byzantine initiator does not send an initial message to any correct process, no correct process will deliver anything. That is because no correct process will send an echo message (then none will send ready messages). Since all advances require $t + 1$ messages from different processes, and Byzantine processes are at most t, the correct processes will be stuck in Step 1, and will make no decision. □

Theorem 4. *Let n be the number of processes, and t be an upper bound of the Byzantine processes. If $n \geq 2t + 1$, in Algorithm 2, the number of TMC-Object used is optimal, in the sense that if we remove the only TMC-Object (initiator), Algorithm 2 does not implement the Byzantine reliable broadcast.*

Proof. In Algorithm 2, only 1 TMC-Object is used, the one at the initiator. If the TMC-Object is removed (instead of doing an TMC-Object-Send, the initiator does only a Send operation), then the algorithm resembles the Bracha Byzantine reliable broadcast protocol [4] where instead of a $2t + 1$ bound to advance, we have only a $t+1$ bound. Since the Bracha Byzantine reliable broadcast protocol is optimal in the number of faults [4,15], Algorithm 2 with no TMC-Object cannot implement Byzantine reliable broadcast. □

6 Conclusion

We focus on Byzantine Reliable Broadcast in trusted execution environments. First we show that adding trusted execution environments to all processes to prevent equivocation does not improve the security threshold or security guarantees of Bracha's Byzantine Reliable Broadcast. Second, we propose an optimal TEE-based algorithm that implements Byzantine Reliable Broadcast in asynchronous settings with resilience $n \geq 2t + 1$. Our algorithm uses a very simple TEE that provides a trusted monotonic counter. Moreover, our solution needs only one trusted monotonic counter at the initiator.

References

1. Beltrando, L., Potop-Butucaru, M., Alfaro, J.: Tendertee: Increasing the resilience of tendermint by using trusted environments. In: 24th International Conference

on Distributed Computing and Networking, ICDCN 2023, Kharagpur, India, January 4-7, 2023. pp. 90–99. ACM (2023). https://doi.org/10.1145/3571306.3571394, https://doi.org/10.1145/3571306.3571394

2. Bonomi, S., Decouchant, J., Farina, G., Rahli, V., Tixeuil, S.: Practical byzantine reliable broadcast on partially connected networks. In: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS). pp. 506–516. IEEE (2021)
3. Bonomi, S., Farina, G., Tixeuil, S.: Reliable broadcast despite mobile byzantine faults. In: Bessani, A., Défago, X., Nakamura, J., Wada, K., Yamauchi, Y. (eds.) 27th International Conference on Principles of Distributed Systems, OPODIS 2023, December 6-8, 2023, Tokyo, Japan. LIPIcs, vol. 286, pp. 18:1–18:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). https://doi.org/10.4230/LIPICS.OPODIS.2023.18, https://doi.org/10.4230/LIPIcs.OPODIS.2023.18
4. Bracha, G.: Asynchronous byzantine agreement protocols. Inf. Comput. **75**(2), 130–143 (1987). https://doi.org/10.1016/0890-5401(87)90054-X
5. Chun, B., Maniatis, P., Shenker, S., Kubiatowicz, J.: Attested append-only memory: making adversaries stick to their word. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007. pp. 189–204. ACM (2007). https://doi.org/10.1145/1294261.1294280
6. Clement, A., Junqueira, F., Kate, A., Rodrigues, R.: On the (limited) power of non-equivocation. In: ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012. pp. 301–308. ACM (2012). https://doi.org/10.1145/2332432.2332490
7. Correia, M., Neves, N.F., Veríssimo, P.: How to tolerate half less one byzantine nodes in practical distributed systems. In: 23rd International Symposium on Reliable Distributed Systems (SRDS 2004), 18-20 October 2004, Florianpolis, Brazil. pp. 174–183. IEEE Computer Society (2004). https://doi.org/10.1109/RELDIS.2004.1353018
8. Correia, M., Neves, N.F., Veríssimo, P.: BFT-TO: intrusion tolerance with less replicas. Comput. J. **56**(6), 693–715 (2013). https://doi.org/10.1093/comjnl/bxs148
9. Decouchant, J., Kozhaya, D., Rahli, V., Yu, J.: DAMYSUS: streamlined BFT consensus leveraging trusted components. In: EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022. pp. 1–16. ACM (2022). https://doi.org/10.1145/3492321.3519568
10. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. In: Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA. pp. 1–7. ACM (1983). https://doi.org/10.1145/588058.588060
11. Guerraoui, R., Komatovic, J., Kuznetsov, P., Pignolet, Y.A., Seredinschi, D.A., Tonkikh, A.: Dynamic byzantine reliable broadcast [technical report]. arXiv preprint arXiv:2001.06271 (2020)
12. Levin, D., Douceur, J.R., Lorch, J.R., Moscibroda, T.: Trinc: Small trusted hardware for large distributed systems. In: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA. pp. 1–14. USENIX Association (2009), http://www.usenix.org/events/nsdi09/tech/full_papers/levin/levin.pdf

13. Maurer, A., Tixeuil, S.: Self-stabilizing byzantine broadcast. In: 2014 IEEE 33rd International Symposium on Reliable Distributed Systems. pp. 152–160. IEEE (2014)
14. Raynal, M.: Fault-tolerant message-passing distributed systems: an algorithmic approach. springer (2018)
15. Raynal, M.: On the versatility of bracha's byzantine reliable broadcast algorithm. Parallel Process. Lett. **31**(3), 2150006:1–2150006:9 (2021). https://doi.org/10.1142/S0129626421500067
16. Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C., Veríssimo, P.: Efficient byzantine fault-tolerance. IEEE Trans. Computers **62**(1), 16–30 (2013). https://doi.org/10.1109/TC.2011.221
17. Yandamuri, S., Abraham, I., Nayak, K., Reiter, M.K.: Communication-efficient BFT protocols using small trusted hardware to tolerate minority corruption. IACR Cryptol. ePrint Arch. p. 184 (2021), `https://eprint.iacr.org/2021/184`