Abstract—State Machine Replication (SMR) is a common technique to make services fault-tolerant. Practical SMR systems tolerate process crashes, but no hardware errors such as bit flips. Still, hardware errors can cause major service outages, and their rate is expected to increase in the future. Current approaches either incur a high overhead by hardening large parts of the system in software, or increase the cost of ownership by introducing additional hardware components.

This work presents HardPaxos, an atomic broadcast algorithm for SMR that enables services to tolerate hardware errors, while incurring little performance and state overhead. HardPaxos requires no additional hardware and has only a small part of its functionality hardened using a combination of AN-encoding and duplicated execution. Our evaluation shows a throughput overhead of at most 5% for typical payload sizes. Moreover, fault injection experiments show that our hardening decreases the number of undetected errors from 15% to 0.02%.

Keywords—hardware errors; Byzantine faults; Paxos

I. INTRODUCTION

State machine replication is a well-known fault tolerance technique, applied in industry-level systems such as Chubby [1], ZooKeeper [2], and Megastore [3]. All these implementations assume a crash-stop failure model, i.e., processes only fail by silently crashing.

Although less frequent than process crashes, hardware errors do occur in production [4], [5] and are expected to happen more often in future hardware generations [6]. Hardware errors can be caused by, for example, aging, cosmic rays, heat and production failures [4]. A non-negligible number of these errors is uncorrectable by hardware techniques such as ECC and manifest to the application as control-flow errors, state corruptions or computation errors. Due to such errors, a process might fail non-silently, sending a corrupted message out and disrupting other processes in the system. In fact, recent events have shown that hardware errors can cause major outages of large distributed systems, for example, the Amazon EC2 incident in 2008 [7].

Byzantine fault-tolerant (BFT) algorithms tolerate not only crashes and hardware errors but even malicious attacks [8]. BFT algorithms are, however, not used in production [9]. BFT systems typically incur higher ownership and maintenance costs than crash-tolerant systems. Instead of $2f + 1$ machines to tolerate $f$ process crashes, BFT systems require at least $3f + 1$ machines to tolerate $f$ arbitrary (Byzantine) failures [8]. Trusted hardware components can reduce the required number of machines [10], [11], but are expensive and slower than commodity hardware [12]. The additional costs are hard to justify if the protection against malicious attacks is an orthogonal concern [13], [14], e.g., in data centers protected by firewalls.

Recently, alternatives to BFT have been proposed for non-malicious environments. For example, Correia et al. [14] and Behrens et al. [15] have presented approaches to harden crash-tolerant systems against hardware errors, having SMR as one use case. The approaches introduce state and execution redundancy, focusing on generic hardening of program code via class inheritance and reflection or via compiler-based transformation. The approaches incur, however, non-negligible overhead due to their generality. An SMR system hardened with the first approach has a 17% to 30% lower throughput than its crash-tolerant counterpart [14]. The second approach incurs even higher overhead with at least 4 times higher latency [15]. Furthermore, both approaches introduce 100% state overhead in the atomic broadcast algorithm, which can hold a large log of requests.

We show in this paper that, by hardening only part of the atomic broadcast algorithm, crash-tolerant SMR systems and the service running on top can tolerate hardware errors, while incurring little state, throughput and latency overhead.

In Section II, we propose HardPaxos, a variant of the Paxos [16]. Paxos is widely used to implement SMR-like systems [1], [3], [17]. In HardPaxos, critical functions and the service running on top can tolerate hardware errors, while incurring little state, throughput and latency overhead.

In Section II, we propose HardPaxos, a variant of the Paxos [16]. Paxos is widely used to implement SMR-like systems [1], [3], [17]. In HardPaxos, critical functions and state variables are factored out of the algorithm and kept in a module called HardCore. In Section III, we describe how to satisfy the Paxos invariants given a trustworthy HardCore.

If the HardCore is an ordinary software module, it is also subject to hardware errors. In Section VI, HardCore is made trustworthy by hardening it in software with AN-encoding [18] and duplicated execution. If a hardware error affects the state or computation of the HardCore, the whole process is aborted with a high probability, i.e., hardware errors are transformed into crashes. Moreover, all messages exchanged in HardPaxos are augmented with end-to-end error detection codes that allow the receiver’s HardCore to detect the incorrect execution of a non-silent faulty process.

In Section VII, we perform extensive fault injection experiments to evaluate the fault coverage of our hardening. The experiments show a decrease of undetected errors from 15% without hardening down to 0.9% with AN-encoding.
and further down to 0.02% with AN-encoding plus duplicated execution. We also show that the hardening does not hamper performance. Even with the strongest hardening, the throughput loss is at most 4% for small messages (64 bytes) and close to zero for messages of 1 KiB or larger; the increase in the single request latency does not exceed 5%.

Finally, Section IV discusses HardPaxos’s correctness, Section V presents the garbage collection support in HardPaxos, and Section VIII presents further related work.

II. RATIONALE

A. System Model

We assume an asynchronous system model with a set of processes \( \Pi = \{ p_1, \ldots, p_n \} \) (with \( n \geq 3 \)) that communicate via message-passing. Each process represents a replica of a service, for example, a database (see Fig. 1). The service runs on top of an SMR library. Clients access the service by issuing requests to one of the processes using a client library. The execution of a replica is given by a sequence of slots; each slot represents one client request. The SMR server library guarantees that correct processes agree on the requests to be executed in each slot. To achieve agreement, the server library employs HardPaxos. HardPaxos contains a trusted module called HardCore, which encapsulates a set of variables and functions described below.

B. Failure Model and Assumption Coverage

In our failure model, messages might be lost, duplicated or reordered, and processes might crash or fail arbitrarily under the restriction of the following assumptions:

A1 There is a bound \( f = \lfloor (n - 1)/2 \rfloor \) on the number of faulty processes; failures occur independently.

A2 Hash functions are collision-resistant and error detection codes cannot be forged.

A3 The trusted module only fails by crashing its process.

For the sake of reasoning about the algorithm, one asserts that these assumptions always hold. In practice, however, assumptions only hold with a probability that depends on the system implementation and deployment environment – this probability is called assumption coverage [19]. For example, Assumption A2 only holds with a high probability if the system is not subject to malicious attacks; this paper focuses on such non-malicious deployments. Assumption A3 only holds with a high probability if either the system is deployed on very reliable hardware – an expensive solution – or if we harden the trusted module against hardware errors (Section VI). Note that A3 does not restrict the failure modes of the untrusted part of the processes, including the service.

Furthermore, we can only provide liveness properties under the following additional assumptions:

A4 Message transmission is eventually timely, i.e., there is a global stabilization time [20].

A5 Faulty replicas eventually become silent, e.g., by crashing, or are rejuvenated.

In practice, after a period of arbitrary behavior, hardware errors often lead to a kernel panic or a machine shutdown – as in the Mortal Byzantine model [21]. Moreover, processes can be periodically rejuvenated [8]. Rejuvenation is, however, an orthogonal concern out of the scope of this paper.

C. HardPaxos Overview

HardPaxos is one possible variant of the Paxos algorithm [16]. Paxos is typically described with proposer, acceptor and learner roles [22]. In HardPaxos, processes play all roles. Hence, we prefer the terms leaders and followers.

HardPaxos satisfies the following properties:

Property 1 (Agreement): If two correct processes \( p_i \) and \( p_j \) deliver requests \( r_i \) and \( r_j \) in slot \( s \), then \( r_i = r_j \).

Property 2 (Validity): Only a request that has been issued by a client may be delivered by correct processes.

Property 3 (Termination): If a client issues a request \( r \), all correct processes eventually deliver \( r \).

During normal execution (see Fig. 2), a distinguished process, called leader, proposes requests issued by clients to all other processes, called followers. The goal of the leader is to get client requests accepted by a majority of processes, e.g., leader itself and half of its followers. A client request comprises the command to be executed by the service, the request’s hash (which identifies and protects the request), a client identification number and a client sequence number (which can be used by the service to provide exactly-once client-order execution semantics). Upon receiving requests from the client, the leader proposes them in slots. Followers accept proposals from the leader by storing them into a log and sending accept messages to the leader – the leader accepts its proposal immediately. Once the leader receives accept messages from a majority of processes, it sends a commit message to the followers and delivers the request to the service. Upon receipt of a commit message, each follower also delivers the request.
The service executes the request and sends a reply to the client via the SMR library. The client library delivers the reply upon receiving $f + 1$ equivalent reply messages from different processes. This voting mechanism [23] is common for BFT algorithms [8] and enables HardPaxos to tolerate hardware errors in the atomic broadcast and in the service.

HardPaxos runs in epochs, each of them having at most one leader. An epoch starts when a candidate demotes the current leader sending a prepare message with a new epoch number to all processes (see Fig. 3). Processes reply to the prepare message with a promise message, promising not to accept any message from older epochs (effectively adopting the proposed epoch); each process sends a log of accepted requests along with the promise. The candidate becomes leader once it receives promises for its epoch from a majority; the candidate immediately receives a promise from itself. The new leader recovers the system by reproposing requests not yet committed. As long as the system is stable, no other process tries to demote the current leader.

We assume messages arrive in FIFO order and are not lost at the level of HardPaxos. That is implemented by the server library using TCP sockets. Additionally, we assume the client library retries a request if no response arrives.

D. HardCore and Certificates

A naive approach to guarantee safety in face of hardware errors would be to move the whole algorithm inside the trusted module. In contrast, our approach only adds enough functionality into HardCore so that the invariants guaranteed by the classical Paxos still hold under our failure model. We keep most of the processing – including communication via sockets, event looping, data structure manipulations, iterations, copying and storing requests, and service on top – outside HardCore to minimize the performance impact incurred by hardening, which can be prohibitive [15].

We illustrate our approach with an example: A leader may propose a request $r$ in a slot $s$ only if no process in a majority has accepted any other request $r'$ in $s$ – otherwise, causing replicas to diverge. Since processes can fail in unpredictable ways, we cannot guarantee a faulty leader never sends different proposals for the same slot $s$. Nevertheless, we can guarantee that only one proposal is legal by requiring proposals to be confirmed by HardCore with a certificate. The leader’s HardCore only confirms a proposal – i.e., creates a certificate – given certificates from a majority of processes declaring that no other request has been accepted in $s$. This approach is pervasive in the design of HardPaxos; each critical action of the protocol is confirmed by HardCore, which verifies input certificates, changes its state, and produces output certificates.

Certificates are enclosed in every message along with the message’s payload, e.g., a proposal consists of a propose certificate and the client request. Certificates are sealed with error detection codes, such that they cannot be forged in the untrusted part of the algorithm or on the network. Upon receipt of a message, correct processes verify the message’s certificate; if the certificate is illegal – i.e., corrupted – the message is discarded. HardCore itself produces illegal certificates if illegal input certificates are given (Section III); or if an internal error is detected (Section VI). Tables I and II describe the certificates of HardPaxos. As an example, a propose certificate consists of the process id $p$, the current epoch $e$, the proposed slot $s$, and the hash $h$ of the request $r$ being proposed. The symbol $\perp$ represents illegal certificates.

III. THE TRUSTED CORE OF HARDPAXOS

In this section we present HardCore. We omit the hardenened part of HardPaxos due to space constraints. A short description was given in Section II-C; the complete algorithm, including message handling and logging, is presented in our technical report [24]. When describing HardCore, we concentrate on why it guarantees the protocol safety. For that we consider two central invariants of Paxos [22].

Invariant 1: A process accepts a proposal in epoch $e$ iff it has not sent a promise in an epoch $e'$ with $e' > e$.

Invariant 2: For each slot $s$, for any request $r$ and epoch $e$, if request $r$ is proposed in epoch $e$, then there is a set $P$ consisting of a majority of processes such that either

(a) no process in $P$ has accepted any proposal for slot $s$ in any epoch $e'$ with $e' < e$; or
(b) request $r$ is the request of the proposal $m$ among all proposals accepted by the processes in $P$ such that $m$ has been issued in the highest epoch $e'$ with $e' < e$.

Algorithms 1, 2 and 3 present the state variables and functions of HardCore. The state mainly consists of three monotonically increasing counters: $E$ counts epochs of the algorithm – called ballot number in Paxos; $A$ counts slots, i.e., the highest slot accepted; $C$ counts the number of commits, i.e., the highest slot known to be accepted by a majority. Other variables are introduced below as needed.

A. Proposing the Right Requests

Invariant 2(a) states the leader can only propose a request $r$ with hash $h$ for a slot $s$ in epoch $e$ if no other request $r'$ has been accepted for the same slot $s$ by a majority in any epoch less than $e$. The Propose() function (Algorithm 1, Line 3) returns a legal propose certificate under two conditions.

<table>
<thead>
<tr>
<th>Certificate</th>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prepare</td>
<td>$p$, $e$</td>
<td>$p$, $e$, $a$, $c$</td>
</tr>
<tr>
<td>Promise</td>
<td>$p$, $e$, $a$, $c$</td>
<td>$p$, $e$, $s$, $h$</td>
</tr>
<tr>
<td>Propose</td>
<td>$p$, $e$, $s$, $h$, $m$</td>
<td>$p$, $s$, $h$</td>
</tr>
<tr>
<td>Commit</td>
<td>$p$, $s$, $h$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>max. accepted slot</td>
</tr>
<tr>
<td>$c$</td>
<td>max. committed slot</td>
</tr>
<tr>
<td>$e$</td>
<td>current epoch</td>
</tr>
<tr>
<td>$a_e$</td>
<td>accepted-at epoch</td>
</tr>
<tr>
<td>$h$</td>
<td>request hash</td>
</tr>
<tr>
<td>$m$</td>
<td>committed mark</td>
</tr>
<tr>
<td>$p$</td>
<td>process id</td>
</tr>
<tr>
<td>$s$</td>
<td>slot number</td>
</tr>
</tbody>
</table>
First, PROMISE certificates from a majority matching the epoch $E$ counter are provided. Second, the certificates have the highest accepted slot $a$ less than $s$, indicating that no process has accepted any other request in $s$ (Line 7). PROMISE certificates mark the highest slot accepted by each process and are retrieved during leader election. Next, the leader sends a proposal with request $r$ and the PROPOSE certificate to all followers; the leader immediately accepts the request using the returned ACCEPT certificate. Note that some functions such as Propose() check whether the number of outstanding accepted requests is within a maximum $K$—an essential check for the garbage collection (Section V).

To ensure Invariant 2(a), Propose() also guarantees that (1) a process can create only one legal PROPOSE certificate for a slot $s$ in an epoch $e$ since the $A$ counter is always incremented; (2) only one process can create legal PROPOSE certificates in an epoch $e$ since leaderOf($e$) = $p$ is true for a single process. As in Paxos, an epoch is a tuple $(i, p)$ of an integer $i$ and a process identifier $p$. Epochs are lexicographically ordered and no two leaders ever choose the same epoch; a process calculates the next epoch increasing the integer and substituting the process identifier with its own.

### B. Election and Accepting the Right Proposals

Before discussing Invariants 1 and 2(b), we explain how recovery works in HardPaxos. When a candidate suspects the current leader is faulty—e.g., using heartbeats—it calls Prepare() in its HardCore (Algorithm 2, Line 3), which in turn increments the epoch counter $E$ and returns a PREPARE and a PROMISE certificate. The candidate adopts the new epoch locally using the PROMISE certificate and sends the PREPARE certificate in prepare messages to all followers.

Upon receiving a prepare, each follower calls Promise() in its HardCore (Line 9). If the PREPARE certificate is legal and the epoch in PREPARE is greater than the current epoch, HardCore saves the epoch counter $E$ in the previous epoch variable $E_p$ and adopts the new epoch $e$ by overwriting its $E$ counter. From this time on, the follower’s HardCore ignores any proposals from older epochs, because the Accept() function only accepts a proposal in epoch $e$ if $e = E$ (Algorithm 1, Line 12). This scheme guarantees Invariant 1.

During recovery, HardCore has to reset the $A$ counter (Algorithm 2, Lines 6 and 13) since a different request might have been accepted on the same slot before a final agreement is reached. A naive approach would be to reset $A$ to 0, but that would make garbage collection impossible (see Section V). Instead, we exploit the fact that slots that have already reached agreement are eventually committed during normal-case execution, and the highest committed slot is marked by $C$. Hence, we reset the acceptance counter $A$ to $C$, and memorize the highest accepted slot in $A_h$.

### C. Promising the Right Accepted Requests

Although not explicit in [22], Invariant 2 only holds if additionally each process only accepts one request per epoch in a slot $s$ and does not forget about any acceptance. The former is easily guaranteed because the acceptance counter $A$ is incremented (Algorithm 1, Lines 5 and 15). The latter requires the Update() function described in Algorithm 3.

For recovery, followers send all accepted (but not garbage collected) requests with their promises to the new leader. These accepted requests along with their ACCEPT certificates are stored in a log kept outside HardCore. Before sending a promise to the leader, each follower binds (updates) each ACCEPT certificate in its log with the current epoch $E$. Outdated certificates are ignored by the leader’s HardCore.

Update() performs two important checks before updating the epoch number of a certificate. First, it checks whether the certificate is bound with the previous epoch $E_p$. The check implies that the unhardened part of HardPaxos has scrupulously updated all certificates for every epoch in which the process has participated. Second, if the slot $s$ has not yet been committed, i.e., mark $m$ is False, then Update() checks whether the slot $s$ has been accepted (or reaccepted) by the
Algorithm 3: HardCore, Leader Election/Recovery for πi

```plaintext
function Update(s, Accepts, Promises) do
if ¬legal(Accept, p, e, e0, s, h, m)) do
  return seal(Accept, p, e, e0, s, h, s ≤ C))
end if
if ¬leaderOf(π) = p, or s ≤ A + 1 or A = C ≥ K
  then return A, :, ;
end if
Vp ← legalSet(Promises);
if |Vp| ≤ f then return A, :, ;
end if
∀p ∈ Vp : v.e ∈ E) ∨ ∀p ∈ Vp : v.a < s then
  return A, :, ;
end if
if ∃v ∈ Vp : v.a ≥ s ∧
  2w ∈ Vw : w.p = v.p ∧ w.s = s ∧ w.e = E then
  return A, :, ;
end if
h ← chooseHash(Va);
A ← A + 1;
return seal((PROPSE, p, A, A, h)),
  seal((ACCEP, p, E, A, h, s ≤ C));
```

process in the previous epoch by comparing it to Ap. In the positive case, e0 must be equal to the previous epoch Ep. In the negative case, the certificate is the latest accepted for the slot since it is bound to the previous epoch Ep, and the same check has been performed before binding it to Ep. The check guarantees that only the latest accepted requests for each slot can be updated. More precisely, Update() ensures that only the ACCEPT certificates for proposals issued in the highest epochs are prolonged (updated) while the elder ACCEPT certificates expire. Finally, the field m of the certificate is marked True if the request has been committed.

D. Reproposing the Right Requests

We now return to Invariant 2(b). The candidate collects all accepted requests and PROMISE certificates from received promises. It becomes leader after receiving legal PROMISE certificates from a majority. The new leader immediately starts the recovery phase: it reproposes every slot not yet committed calling Repropose() with the ACCEPT certificates for s and the PROMISE certificates for the current epoch. Algorithm 3, Line 5, describes the Repropose() function. It starts by checking whether the process is the leader for the epoch and whether the slot reproposed is the next slot. It then checks if there is a majority of legal, bound to the current epoch PROMISE certificates. HardCore also checks that at least one of the PROMISE certificates has a slot greater or equal to slot s since s cannot be reproposed if it has never been proposed before. Next, HardCore selects the legal ACCEPT certificates out of Accepts. Line 13 checks that every process that accepted no less than s requests has an ACCEPT certificate. Processes that do not provide ACCEPT must instead provide a PROMISE certificate with field a less than s; this way they prove they have never seen any request in s. Line 16 selects the hash h from the ACCEPT certificate with the highest accepted epoch e0 just as in Paxos. A faulty process could call Repropose() with no ACCEPT certificate for slot s, but then HardCore returns an illegal certificate.

As a result, HardCore ensures Invariant 2(b) holds because (1) processes cannot lie about what they accepted in a slot; (2) only the latest accepted requests are prolonged from epoch to epoch; (3) there is only one process reproposing in an epoch; (4) the leader can repropose only one request per epoch per slot; and (5) the leader can only repropose the request of the proposal issued in the highest epoch.

IV. HardPaxos Correctness

We now briefly show how the properties of Section II-C are satisfied by HardPaxos.

1) Agreement: From Invariants 1 and 2 follows that, once a majority of processes agrees on a request r in a slot s, all subsequent leaders propose r for s. When the current leader gathers ACCEPT certificates from a majority, HardCore generates a COMMIT certificate. Correct processes only deliver requests upon receiving a COMMIT certificate. Hence, no two correct processes ever deliver different requests in the same slot. A faulty process can still deliver a different request, but the reply will be filtered by the client library.

2) Validity: Clients protect their requests with a hash, which is checked by the service before executing. HardPaxos satisfies validity since error correcting codes cannot be forged by assumption. A leader cannot create a request out of the blue and calculate its hash correctly only due to hardware errors. Assumption A2 allows the implementation to use a simple CRC function to protect the client requests.

3) Termination: Termination is guaranteed if eventually links are stable and faulty processes become silent (A4 and A5). Without assuming that faulty processes eventually become silent, a “babbling idiot” could hamper progress by, for example, constantly sending ever increasing PREPARE messages; or by simply flooding the network switches.

V. Garbage Collection

In practical systems, the log of accepted requests cannot grow without bounds. This section explains how HardPaxos garbage-collects obsolete log messages.

Garbage collection employs three mechanisms. First, the SMR server library asks the service to snapshot its state every Z commits. Z is called snapshot period. Second, we tie together acceptance and commit counters by allowing the A counter be incremented only if C ≥ A − K, where K is a constant called commit distance. Consider, for example, K = 20 and Z = 100. K guarantees that if A = 121 on process p, then p did a snapshot at slot 100 since C has to be at least 100 to increment A from 120 to 121.

The third mechanism is snapshot slots – slots for which A is such that (A − K) modulo Z is 1. After proposing 120 slots, the leader constructs the next proposal with A = 121
in a special way: the proposal contains no payload, and its hash value \( h \) is assigned the digest of its latest snapshot (at slot 100). When followers receive this proposal, they compare the received digest with the digest of their own snapshot at slot 100, and reply with an accept message only if the digests are the same. After the leader receives accept messages for slot 121 from a majority, it commits, making the snapshot stable. The leader then sends a commit message to the followers with \( C = 121 \). Once a process receives the commit message, it can prune its log up to slot \( s = 100 \).

The snapshot certificate – i.e., a commit certificate for a snapshot slot – guarantees that \( f + 1 \) processes agreed on the snapshot. Therefore, at least one correct process made the snapshot. Since this certificate contains not only the slot, but also a snapshot digest, the certificate alone is sufficient to fetch a correct snapshot from another process.

When a process \( p \) starts lagging behind, \( p \) performs a “catch up” to bring itself up-to-date. When \( p \) receives a proposal with slot higher than its \( A+K \), it fetches the latest snapshot certificate (from a majority) and the snapshot itself from any process. The snapshot can be applied to \( p \)'s state if snapshot's digest matches the latest snapshot certificate. Moreover, \( p \) calls a special CatchUp() function in HardCore, which updates the \( C \) and \( A \) counters to the slot of the snapshot certificate. Finally, \( p \) retrieves the latest proposals and commits from the leader as usual in Paxos.

VI. ENFORCING TRUST IN HARDPAXOS

HardPaxos only satisfies Agreement if the HardCore never fails arbitrarily (Assumption A3), i.e., computation and memory errors affecting the integrity of the variables inside HardCore can only result in detectable benign failures such as illegal certificates or process crashes. We enforce A3 with the AN-encoding technique, a hardening technique that protects programs against transient and permanent hardware errors with high probability [25]. Moreover, we combine AN-encoding with duplicated execution to improve fault coverage. We first introduce AN-encoding and its error model and then describe how we harden HardCore.

A. AN-encoding and Error Model

AN-encoding extends the representation of each variable \( x \) of a program module from \( n \) to \( n + k \) bits such that only \( 2^n \) of the \( 2^{n+k} \) values are code values, while the remaining are non-code, i.e., invalid. In practice, AN-encoding transforms, for example, 32-bit variables into 64-bit variables and encodes the functional value \( x_f \) of \( x \) with a constant \( A \) by multiplying it: \( x_c = x_f \cdot A \). The domain of \( x \) is divided into a few code values (multiples of \( A \)) and many non-code values (not multiples). The integrity of \( x \) is verified applying the modulus with the constant \( A \). If the result is zero, \( x \) is code, otherwise non-code. A code value \( x_c \) affected by random error is non-code with a high probability, approximately \( 1 - \frac{1}{A} \) [25]. Moreover, operations are also transformed, so that correctly executed encoded operations preserve the code, and incorrectly executed encoded operations break the code with high probability [18].

Forin [25] classifies state corruptions and computation errors in symptoms perceived at the program level. In this model, AN-encoding can detect faulty operations, i.e., incorrectly executed operations, and modified operands, e.g., variables corrupted by bit flips or stuck-at bits. AN-encoding cannot detect some error classes: exchanged operator, e.g., an addition is exchanged by a subtraction; exchanged operand, e.g., a variable argument is exchanged with another variable due to a corrupted address; and lost updates, e.g., a value is stored in the wrong address and the correct address keeps its old (possibly code) value. Moreover, control-flow errors are only detected if variables become non-code.

B. Hardening HardCore

We employ an AN-encoding compiler\(^1\) for C code. During run-time, if a non-code value is detected, the process is aborted, transforming a hardware error into a crash failure.

1) Overhead: AN-encoding is known to incur high execution and state overhead [18]. Nevertheless, relative to the whole process, which includes SMR library and service, AN-encoding has a limited performance impact (Section VII). Similarly, the state overhead is rather small, doubling the HardCore’s state size from 28 to 56 bytes and certificates’ size, for example, from 24 bytes to 48 bytes for accept.

2) End-to-end Protection: Certificates are always kept encoded, i.e., during computation inside HardCore and also during storage and transmission outside HardCore. Since the certificates are always kept encoded, the function seal() in Section III simply returns the certificate – no additional error detection codes have to be used other than AN-encoding itself. The function legal() in Section III returns True if every encoded value in the certificate is a code value.

3) Improved Fault Coverage: To improve the probability of detecting control-flow, exchanged operator and lost update errors, we duplicate the HardCore module. Every call to HardCore is executed by both copies of the module and certificates are compared on return. If an error is not detected by AN-encoding, a discrepancy on the returned certificates should be seen as long as the fault affects only one of the copies. To provide end-to-end protection, the final certificate contains the certificates from both HardCore copies. In Section VII, we evaluate both approaches: AN-encoded and duplicated-AN-encoded.

VII. EXPERIMENTAL EVALUATION

A. Performance Evaluation

We focus on the questions: (1) What is the hardening overhead of HardCore? (2) What is the latency overhead perceived by the clients? (3) Can HardPaxos reach the same throughput as a crash-tolerant SMR library?

\(^1\)http://silistra-systems.com
1) Implementation and Baselines: We have implemented an SMR library in C\(^2\). In the following, we give the library the same name as the algorithm – HardPaxos. HardPaxos uses TCP sockets to communicate with clients and between replica processes. HardPaxos employs an adaptive batching similar to PBFT’s: if a maximum number of infight proposals is reached, the following requests are batched in a single proposal. The batch size is limited to 30 requests.

HardPaxos comes in two flavors: AN and 2AN. Whereas AN has HardCore hardened with AN-encoding, 2AN has HardCore additionally hardened with duplication execution. We compare HardPaxos against Paxos, a crash-tolerant, performance upper-bound baseline; and against PBFT\(^3\), a Byzantine fault-tolerant, performance lower-bound baseline.

In Paxos and, consequently, HardPaxos, the leader could quickly become a network bottleneck: for each request \(r\), the leader proposes \(r\)’s payload to the followers (Prepare in Fig. 2) and sends the reply to the client after \(r\) has been executed (Reply in Fig. 2). To unload the leader’s channels, the leader does not send a reply to the client; only the followers provide reply messages. The client library waits for at least \(f + 1\) replies with the same payload, and in case some follower fails to send a reply, the client retries.

2) Setup: Experiments were conducted on a cluster of 30 machines with 2 Intel Xeon E5405 2.0 GHz processors, 8 GiB of RAM, connected with Gigabit Ethernet, and running Linux 3.8. The maximal TCP bandwidth is 118 MiB/s.

We assume ideal conditions: links are up and timely, and there are no other jobs running on the machines. All protocols are configured to tolerate 1 fault, i.e., we use 4 replicas for PBFT and 3 replicas for Paxos, HardPaxos-AN, and HardPaxos-2AN. Replicas run on dedicated machines, and clients are distributed uniformly among the other machines of the cluster. Latencies are computed for each client request, and an average is calculated every second. Throughput is controlled by increasing the number of clients. PBFT has disabled multicast (our cluster does not support it).

3) HardCore’s Performance: We start by measuring the overhead incurred by hardening in our HardCore module. We create a micro-benchmark that mimics the behavior of a leader and a follower in a loop. Table III shows the number of hardware cycles per commit for native HardCore, i.e., unhardened, and HardCore hardened with AN and 2AN. The difference between AN and 2AN is twofold since 2AN essentially runs the same code twice using two state copies. As expected, the overhead in comparison to native is high: about 39 times for leader and 27 times for follower. Hence, encoding would incur a prohibitive overhead if applied to the whole process. Yet, as we show next, this overhead is not prohibitive in the context of HardPaxos.

4) Latency-Throughput Results: To evaluate the hardening overhead in HardPaxos, we use an echo service benchmark. The echo service receives a client request with a dummy payload and sends back a reply with the same payload. We experiment with payloads of 64, 1024, and 4096 bytes. We focus on this benchmark because it shows the overhead of hardening in the atomic broadcast protocol. Remember that the service is not hardened, only the HardCore. Our experiments measure throughput and latency in graceful runs of PBFT, Paxos, and the two versions of HardPaxos.

When clients send small messages, the throughput is limited by the leader’s CPU (Fig. 4). The utilization reaches 100% for all 4 protocols at high loads. HardPaxos has a difference to Paxos of about 3-4%. For example, at 10 ms latency, both HardPaxos-AN and HardPaxos-2AN reach 96% of Paxos’ throughput. Moreover, HardPaxos shows at least 2.1 times higher throughput than PBFT. Hence, hardening only part of HardPaxos incurs an acceptable overhead.

PBFT has a lower latency than Paxos and HardPaxos up to 38 k.op/s because (a) PBFT uses UDP protocol for all communication whereas Paxos and HardPaxos use TCP; (b) to minimize implementation efforts, our Paxos and HardPaxos libraries use memcpy() and queues in several places in the leader’s critical path, introducing a higher overhead than PBFT’s optimized implementation. Moreover, due to our batching implementation, the latency of Paxos and HardPaxos is higher, the smaller the message size is.

Fig. 5 shows experiments with payloads of 1 KiB, the typical payload size to evaluate systems such as ZooKeeper [2], [14]. Already with 1 KiB payloads, HardPaxos variants achieve the network limit before reaching the CPU limit. In the worst case, HardPaxos-AN presents 620 \(\mu\)s higher latency than Paxos, and HardPaxos-2AN 1,230 \(\mu\)s. This is due to higher CPU utilization, which in its turn is induced by the additional CRC computation and hardening of HardCore. At high loads, however, both achieve the limit of 50 k.op/s; here HardPaxos and Paxos show the same throughput and latency due to aggressive batching. The maximal throughput achieved by HardPaxos is about 2.3 times higher than PBFT.

Fig. 6 shows that, regardless of the hardening, HardPaxos achieves the same throughput as Paxos with large messages. Fig. 5 and 6 show HardPaxos and Paxos saturate the network at about 50 MiB/s of payload. Since the leader sends requests to two followers, only half of the bandwidth is available.

5) Single Request Latency: Table IV shows the mean latency for single requests of one client and payload sizes of 64 B, 1 KiB, and 4 KiB. PBFT is slightly faster than Paxos for all cases. HardPaxos-AN and HardPaxos-2AN present
about 4% higher latency than Paxos, which demonstrates the low overhead incurred by hardening under low loads.

B. Fault Injection

We now evaluate the fault coverage of HardCore, considering the questions: (1) With no hardening, is the number of arbitrary failures caused by injected faults non-negligible? (2) Can HardCore with AN-encoding detect as many errors as HardCore with AN-encoding and duplicated execution? (3) What errors can cause the hardened HardCore to fail?

1) Methodology: To study the fault coverage of HardCore, we wrote a simple test case in three variants: native, AN and 2AN. The test case calls the most used functions of HardCore: Propose(), Accept(), Commit() and Committed(). The functions are called with pre-defined arguments, and the HardCore’s state and certificates are checked on return. For example, after a call to Propose(), the test checks whether: (1) the returned PROPOSE and ACCEPT certificates are legal, (2) the PROPOSE and ACCEPT certificates have expected values, and (3) the HardCore’s state has expected values.

We inject one fault in one function for each run. The outcome of a run is classified as: Crash, if it crashes due to segmentation faults or assertions; Hang, if it did not return after 10 seconds; Return ⊥, if the function returns an error code; AN-crash, if the run crashes due to a hardening detection; Illegal certificate, if the function returns a non-code certificate (i.e., detectable with AN checks); Illegal state, if the HardCore’s state is non-code; Unexpected certificate or unexpected state, if a certificate or the state is code but unexpected (i.e., not detectable). The latter two failures are severe: they might propagate to other components, breaking safety properties and bringing replicas to inconsistent states.

To inject faults, we wrote a plugin for Pin tool⁴. The plugin randomly selects an instruction of the target function to inject one fault. For each of the four functions, the test runs as many times as to gather 1,000 runs with detected outcome. For each injection, the plugin randomly selects one of the following low-level faults: CF (corrupts instruction pointer), WV (corrupts memory location after being written), RV (corrupts memory location before being read), WA (corrupts address before writing to it), RA (corrupts address before reading from it), WR (corrupts register after being written), or RR (corrupts register before being read). A corruption randomly modifies a register or an address.

2) Results: Table V summarizes the runs that resulted in some form of failure. Native variant failed in 4688 out of 9466 runs; AN failed in 4036 out of 6216 runs; and 2AN failed in 4001 out of 6143 runs. For all variants, most of the failures are detections caused by segmentation faults or assertions. Almost 15% of the failures of the native (unhardened) HardCore were undetected, i.e., could compromise the safety of the system. With AN-encoding 0.9% of the failures were undetected, while with AN-encoding and duplicated execution only 0.02% (numbers in parenthesis are the absolute number of runs). Although most of the hardening detections stopped the process (AN-crash), 0.4% of them were detectable non-silent failures.

3) Undetected Errors: To understand the causes of undetected errors in the hardened variants, Table VI shows the absolute number of undetected errors that caused failures for each fault type. When possible, we relate the injected faults with Forin’s high-level error model (Section VI).

Because they modify addresses, WA faults are likely to result in lost updates, which are undetectable by AN-encoding. We analyzed the log files and the disassembled binary to find out the symptoms of the remainder fault types. To our surprise, most undetected errors were caused by faults

⁴http://www.pintool.org
that lead to lost updates. For example, in one CF case the program jumped from the preamble of a store function into the preamble of a function that does an overflow check. Once the function returned, the caller continued execution without noticing the missed store. Moreover, 8 out of 10 RV cases were lost updates: some pointer was modified before being loaded into a register and then used to store a value.

Other than lost updates, we found one case of exchanged operand (not detectable by AN): The single RA case was an addition which had the address of one of its operands modified to point to another code word in the memory. One of the RV cases was a pointer corruption that modified the HardCore’s state. When calling Propose(), the implementation passes two pre-allocated pointers which hold the return certificates. Before being encoded, one of them was corrupted by an offset inside HardCore’s state. When Propose() wrote into the certificate, it overwrote the HardCore’s state.

The single case of HardCore-2AN seems to be a false positive: the pointer of the return certificate passed to Propose() was corrupted on the stack of Propose() before being encoded. Both executions of the HardCore wrote into the wrong memory. Once Propose() returned, the pointer in the test case was still correct, but its content was an old certificate. Old certificates are promptly rejected by the receiver’s HardCore; hence, they are benign faults.

**VIII. Related Work**

Companies like Google, Yahoo! and Microsoft successfully employed SMR in their systems [1], [2], [3], basing their implementations on the crash-stop fault model.

Recently, several resource-efficient BFT algorithms have been published such as A2MPBFT-EA [26], MinBFT [11] and CheapBFT [10]. Although all these protocols rely on trusted modules, HardPaxos differs from them in several ways. First, these protocols are based on PBFT by Castro and Liskov [8], which tolerates Byzantine failures, whereas our work is based on Paxos [16], which is crash-tolerant. HardPaxos sends less messages than PBFT-based algorithms because it does not broadcast accept and commit messages, and has a simpler leader election and recovery. Second, they require tamper-proof trusted modules since they consider malicious adversaries. A2M trusted module is implemented either in a trusted VM or as a hardware component; MinBFT’s trusted module is implemented using a Trusted Platform Module (TPM); and CheapBFT’s trusted module is built on FPGA. Third, these works assume the trusted module does not fail arbitrarily. In our work, we enforce the HardCore to detect hardware errors with a high probability.

Our trusted module is more complex than MinBFT’s and CheapBFT’s, but simpler than A2M since it only contains counters. Notwithstanding, our approach could be used to harden a software-only version of their trusted modules.

A well-known software-based hardware-fault tolerance solution is SWIFT [27]. SWIFT is a compiler-based technique to detect single bit flips. The main approach is to repeat the execution of every instruction computing new values, and to subsequently compare these values. SWIFT does not focus on end-to-end protection in distributed systems; it treats any operation that writes into or reads from memory as an I/O operation. In contrast, PASC [14] guarantees error isolation for distributed systems: either the faulty process itself detects the error and crashes, or the recipient detects a faulty message and drops it. Hardening is achieved by duplicating the state and the computation, as well as the message contents. Since PASC duplicates the computation of the whole process, the execution overhead is higher than with our approach: the state overhead incurred by PASC is 100%. We also duplicate our trusted module to increase its error detection, but since HardCore is only a small component in relation to the whole HardPaxos library, we have little state and execution overhead. Moreover, PASC can only detect transient errors, while our approach detects permanent errors by employing AN-encoding.

We have used AN-encoding to harden the HardCore. AN-encoding is can be implemented in hardware [25] or in software [18]. AN-encoding has already been used in distributed systems [15] to achieve error isolation. The approach, however, suffers from even higher execution overhead than PASC as the whole process has to be encoded. In contrast, we have encoded only a small part of HardPaxos.

**IX. Conclusion**

Hardware errors are a threat to distributed systems that can have catastrophic consequences. Because hardware errors occur less often than process crashes, a solution to tolerate them should incur as little overhead as possible. In this paper, we have presented HardPaxos, an atomic broadcast protocol, and the corresponding State Machine Replication library. By hardening only a small portion of HardPaxos with software techniques, one can enable crash-tolerant SMR systems to additionally tolerate hardware errors.

HardPaxos is a hardened variant of the industry-standard Paxos algorithm and presents excellent fault coverage at small performance costs. In the context of SMR systems, we believe HardPaxos to be more practical against hardware errors than existing approaches in the literature such as BFT [8] or hardening the complete processes [14], [15].

**Acknowledgment**

This work was partially supported by the state of Saxony under grant of ESF 100111037 (SREX), by DFG under...
grant FE 1035/1-2, and by the German Excellence Initiative, Center for Advancing Electronics Dresden (cfAED), Resilience Path.

REFERENCES


