Practical Hardening of Crash-Tolerant Systems

Miguel Correia
IST-UTL / INESC-ID
Lisbon, Portugal

Daniel Gómez Ferro
Yahoo! Research
Barcelona, Spain

Flavio P. Junqueira
Yahoo! Research
Barcelona, Spain

Marco Serafini
Yahoo! Research
Barcelona, Spain

miguel.p.correia@ist.utl.pt, {danielgf, fpj, serafini}@yahoo-inc.com

Abstract

Recent failures of production systems have highlighted the importance of tolerating faults beyond crashes. The industry has so far addressed this problem by hardening crash-tolerant systems with *ad hoc* error detection checks, potentially overlooking critical fault scenarios. We propose a generic and principled hardening technique for Arbitrary State Corruption (ASC) faults, which specifically model the effects of realistic data corruptions on distributed processes. Hardening does not require the use of trusted components or the replication of the process over multiple physical servers. We implemented a wrapper library to transparently harden distributed processes. To exercise our library and evaluate our technique, we obtained ASC-tolerant versions of Paxos, of a subset of the ZooKeeper API, and of an eventually consistent storage by implementing crash-tolerant protocols and automatically hardening them using our library. Our evaluation shows that the throughput of our ASC-hardened state machine replication outperforms its Byzantine-tolerant counterpart by up to 70%.

1 Introduction

Distributed systems in production require dependability mechanisms to avoid extended periods of unavailability, violations of data integrity, and other undesirable consequences of faults. Coordination systems, such as ZooKeeper [24], and storage systems, such as GFS [18] and Bigtable [11], all include mechanisms to prevent faults from disrupting their operation. These systems are all designed to tolerate crashes, since crashes are observable and are common in production environments. Crash-tolerance assumes that processes fail in a silent manner and never send incorrect messages. This assumption, however, may be violated in presence of undetected corruptions of the internal state of a faulty process. The impact or even the sheer occurrence of such data corruptions can be very hard to ascertain because they are not properly detected by the system. One infamous failure occurred in the Amazon S3 storage service in July 2008 [2]. It required taking the whole system down for an incremental restart and resulted in an 8-hour outage. Post-mortem failure diagnosis concluded that:

> *A handful of messages had a single bit corrupted such that the message was still intelligible, but the system state information was incorrect. We used MD5 checksums throughout the system (but not) for this particular internal state information. (...) When the corruption occurred, we did not detect it and it spread throughout the system causing the symptoms described above [2].*

Other failures have been traced back, or hint to, corruptions of the internal state of a process; for example, replicas diverging in the Chubby lock service [10], data loss in Ma.gnolia [31], and data corruption in S3 [3]. Publicly known failures are often related to services exposed to external users. The scale of the problem possibly goes far beyond these known cases, since companies often keep data corruption incidents confidential.

Practical distributed systems often use CRCs, MD5 hashes, or other error detection codes to detect data corruptions. Due to the lack of viable principled approaches, however, developers often find it difficult to reason about appropriate placements of checks into their code. *Ad hoc* error detection might not cover important fault scenarios, making the system susceptible to severe outages.

In this paper, we propose a principled hardening approach that specifically targets *realistic* faults in distributed systems. We surveyed faults that have been observed in post-mortem analysis of production datacenter systems or through fault injection campaigns. We then focused on the ones that can be tolerated through distributed fault tolerance techniques, which comprise many of the faults we surveyed. We observed that these faults manifest at the process level as state corruptions or control-flow corruptions, rather than as "adversarial" process behavior. Our fault injection experiments on Paxos confirm this observation.

Based on the above observations, we propose a new fault model for Arbitrary State Corruption (ASC) faults, modeling the effect of realistic faults at the level of a process of a distributed system. ASC admits that faults change the state of a faulty process to an arbitrary value and modify the execution flow of the process code. ASC faults can occur an unbounded number of times.

We then propose a *hardening* technique that guarantees error isolation: an ASC-faulty process does not propagate erroneous state to other processes through erroneous messages. Either the faulty process itself detects the error and crashes, or the recipient detects a faulty message and drops it. The designer of a distributed protocol can thus focus on the tolerance of crashes and message omissions, which are simpler to handle, and is relieved from the burden of applying error detection checks and reasoning about guarantees when introducing them. The hardened version of a process is still a single process, which does not need to be replicated over multiple physical machines. Hardening is achieved by adding redundancy to the state and the computation of the original process, as well as to the messages it sends. No trusted component is assumed: the hardening state and computation can also be corrupted.

We automated our hardening technique by designing a library, called PASC, that wraps processes and transparently hardens them. Our hardening algorithm only assumes that processes communicate with each other through message-passing. Using PASC, we obtained ASC-tolerant versions of protocols with different consistency, fault tolerance, and scalability properties, simply by hardening the processes of corresponding crash-tolerant protocols, with only minor modifications. We implemented the Paxos protocol [28], a subset of the ZooKeeper API on top of Paxos, and a scalable eventually consistent storage, with acceptable performance overhead (around 17% less throughput for ZooKeeper).

A safer but more expensive alternative to our approach is tolerating Byzantine faults, where faulty processes can turn into adversaries and make any theoretically possible action to harm other correct processes. Byzantine-fault tolerance (BFT) protocols implement strongly consistent state machine replication (SMR). Beyond data corruptions, BFT tolerates software bugs, intrusions and other malicious process behavior under the assumption that a quorum of correct replicas is always available. This can be achieved by using diverse replicas [17], but development costs and the difficulty of achieving independence of failures in practice [26] prevent the use of design diversity in many cases. Despite the good performance of existing BFT algorithms [9, 27, 39, 43], and the existence of prototypes of complex systems using BFT [1], the industry has not adopted BFT as a viable solution to dependable systems, to the best of our knowledge.

ASC-hardening differs from BFT in a number of key aspects. First, it considers security orthogonal to fault tolerance, which is what most practical systems do, and focuses on the latter. Second, it does not need replication to prevent error propagation. Existing approaches preventing error propagation when processes fail in a Byzantine manner use variants of SMR, resulting in higher costs [23]; furthermore, many distributed systems do not need replication of all their processes, or use replication with weaker consistency than strongly consistent SMR. The third key difference is that any process of a distributed system can be ASC-hardened, including clients of client-server and multi-tier architectures; this is an important property in practical distributed systems [22], but achieving it with a Byzantine fault model can be very complex [33]. Finally, ASC-hardening is significantly more efficient than BFT. Our micro-benchmarks show that ASC-tolerant Paxos delivers up to 70% higher throughput than the BFT protocol of Castro and Liskov [9].

In this paper we make five main contributions:

- A new process-level fault model, ASC, representing realistic data corruptions;
- A sound technique to harden crash-tolerant processes against ASC faults;
- A library, PASC, that enables developers to automatically harden crash-tolerant protocols;
- An evaluation of ASC-hardened versions of Paxos, of a subset of the ZooKeeper API, and of an eventually consistent store;
- A validation of the ASC model and of the coverage of PASC by injecting code and state corruptions in our Paxos implementation.

2 Realistic faults

Understanding failure behavior of computer systems has been the topic of decades of research, which traditionally focused on monolithic mainframe systems (see for example the work by Jim Gray on the Tandem system [19]). We surveyed recent failures of distributed production systems, which are often built using commodity and low-end servers. We also included results of large-scale studies on faults in such systems. We did not consider security issues because, in most production system, they are not handled using fault tolerance techniques.

We selected representative reports that are publicly available; it is not our intention to provide an exhaustive or comprehensive survey of all failure events in production systems. This discussion, however, is sufficient

N.	Failure description	Class of faults	Crash	ASC	Byz.
1	Bugs in several database management systems with different designs shown to corrupt data in study [17]	Software development faults	no		yes †
2	Google App Engine unavailable due to a bug in the GFS Master: a malformed file handle caused successive crashes and recoveries (July 9 2009) *(http://groups.google.com/group/google-appengine/msg/ba95ded980c8c179)*	Software development faults	no		maybe †
3	Paxos replica state corruption due to an illegal memory access made by errant code included in the codebase [10]	Software development faults	no		maybe †
4	Two Yahoo! client applications had bugs caused by developers that misinterpreted the semantics of the ZooKeeper API [40]	Software development faults	no		
5	Administrators configured Nagios to monitor ZooKeeper by pinging certain TCP ports; each ping created a thread that was not closed, leading to too many threads *(https://issues.apache.org/jira/browse/ZOOKEEPER-880)*	Operation faults	no		
6	DNS misconfiguration caused machine names to clash, preventing a ZooKeeper instance running at Yahoo! from continuing to work (July 2009) [40]	Operation faults	no		
7	Incorrect NIC configurations caused frequent leader elections and system instability in a ZooKeeper instance running at Yahoo! [40]	Operation faults	no		
8	Google App Engine datastore unavailability due to instability in the cluster, which overloaded the Bigtable repository keeping the locations of chunks; timeouts started to expire and all requests started to fail (May 25 2010) *(http://groups.google.com/group/google-appengine-downtime-notify/msg/e9414ee6493da6fb)*	Operation faults	no		
9	Facebook offline for 2.5 hours due to automatic error correction mechanism that overloaded a database with queries (September 23 2010) *(http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919)*	Operation or Software development faults	no		
10	Paxos replica inconsistency despite consistent execution logs, probably caused by a hardware memory error corrupting the replica state [10]	Hardware fault	no	yes	
11	Uncorrectable corruption of data in ECC DRAMs shown to happen in large-scale study [38]	Hardware faults	no	yes	
12	Amazon S3: A new but defective load balancer corrupts some relayed messages until it is taken offline, after 36 hours (June 20-22, 2008) *(https://forums.aws.amazon.com/thread.jspa?threadID=22709)*	Hardware faults	no	yes	
13	Amazon S3 unavailability due to corruption of a single bit in a few messages, which propagated errors to multiple servers; this generated a high volume of gossiping of the system state, bringing request processing close to a halt (July 20, 2008) *(http://status.aws.amazon.com/s3-20080720.html)*	Hardware faults	no	yes	
14	Sun servers fail randomly due data corruption in memory caches caused by cosmic rays [44]	Hardware faults	no	yes	
15	Ma.gnolia looses half terabyte of its customers' data due to file system corruption (Jan. 09) *(http://getsatisfaction.com/magnolia/topics/ ma_gnolia_data_recovery_status)*	Hardware faults	no	yes	
16	Sidekick unavailability, apparently due to disk failure; the data took several days to be reconstructed (October 2009) *(http://latimesblogs.latimes.com/technology/2009/10/microsoft-says-lost-sidekick-data-will-be-restored-to-users.html)*	Hardware faults	no	yes	
17	Undetected data corruptions in disks, often due to buggy firmware, shown to happen in large-scale study [5]	Hardware faults	no	yes	
18	SSD devices found to lose data due to incorrect firmware (2009) *(http://www.dailytech.com/Update+Intel+Confirms+SSD+Data+Corruption+Issue+Suspends +Shipments+Pending+Firmware+Update/article15827.htm)*	Hardware faults	no	yes	
19	IOMMU chipset of some AMD boards undetectably corrupts data due to a bug, which is only activated in some specific hardware/software configurations *(https://bugzilla.kernel.org/show_ bug.cgi?id=7768)*	Hardware faults	no	yes	
20	Google services partially unavailable for approximately 1 hour due to fire in a datacenter (2010) *(http://mobilelocalsocial.com/2010/google-data-center-fire-returns-world-wide-404-errors/)*	Hardware faults	yes ⋆		
21	Los Angeles International Airport law enforcement database unavailable due to a faulty NIC that caused a packet storm and took the entire LAN offline (August 2007) *(http://www.crn.com/news/networking/201801022/nic-card-soup-gives-lax-a-tummy-ache.htm)*	Hardware faults	yes ⋆		

Table 1: Recent failure events and large-scale dependability studies. In several cases there was a cascade of faults, so the class refers to the root cause. The three columns on the right indicate the ability of distributed crash, ASC and Byzantine fault-tolerant systems to tolerate the faults that caused the events († these faults are tolerated only if replicas fail independently, e.g. through design diversity; ⋆ only using multiple networks/locations).

to illustrate that real failures can be quite complex, with process faults that are more severe than just crashes.

Failure reports. Table 1 loosely organizes failure events and studies in classes. In some cases the assignment to a class is speculative due to lack of detailed information.

Software development faults, or bugs, are a pervasive class of faults (rows 1-4 in the table). Even in well-tested software, bugs whose activation depends on intricate combinations of states and inputs and/or heavy loads are usually present [4] and can lead to data corruption in databases (rows 1, 3) or cause effects like cyclic crashes and recoveries (row 2). Bugs cannot be tolerated using commonly available distributed fault-tolerant techniques. In principle, some development faults (row 1) may be tolerated using BFT if processes are replicated and each replica contains different software, so the bug does not affect all of them [17]. However the use of diversified design is often not feasible.

Some of the faults of Table 1 cannot be tolerated even with BFT. For example, operation faults are human errors made during system configuration and at runtime. Row 5 shows a case in which a monitoring tool was configured in a way that was not predicted by the software developers, leading to the consumption of threads, eventually causing abnormal operation. Several other examples are shown in rows 6-9. These faults are hard to tolerate automatically regardless of the fault model adopted because they can easily result in correlated failures ("complex human-machine interactions (...) remain a challenge" [4]). Client-side software development faults (row 4) or bugs in the implementation of trusted fault tolerance libraries (row 9) are also not tolerated by BFT.

Despite these inherent limitations, there is a large fraction of faults that can be handled using distributed fault tolerance. Hardware faults are known to cause corruption of data in RAM (rows 10-13), memory caches (row 14), and hard disks (rows 15-17). Some hardware faults are actually caused by development faults in hardware components or, more often, in hardware-related software such as firmware or drivers. It is well known that firmware can cause data corruptions in hard disks; such corruptions can only be detected using additional checks at the operating system or application level (row 17). Other classes of block storage devices, like solid-state drives (SSD), may also corrupt data (row 18). Most current servers use Single Error Correction, Double Error Detection (SECDED) ECC DRAMs, but a recent large-scale study has found that their error rates are orders of magnitude higher than previously reported and that several uncorrectable errors are detected (row 16). It is not possible to know how often undetected memory errors violate data integrity without being reported; some analytical model indicates that such errors may be not uncommon [14]. The study reports a correlation between CPU load and rate of detected memory errors, indicating that data corruptions may also occur in the data path to and from memory. For example, motherboard development faults have been found to cause undetected data corruptions (row 19).

Data-center wide failures due to environmental reasons such as fires (row 20) have also caused massive outages. Network failures are known to corrupt the content of messages, but sometimes may make large networks unavailable (row 21).

After reviewing these failure reports, we conducted a closer analysis of realistic faults whose effects can be tolerated using BFT without design diversity, as for example hardware errors. Our goal was understanding how these faults can be observed by processes of a distributed system. There are three main observations we could draw from this analysis.

Observation 1: State corruptions cause most failures. The tolerable failures of Table 1 are caused by corruptions of the *state* of some process rather than arbitrary, or even adversarial, process behaviors. This is consistent with recent studies that have investigated in detail the effect of systematic injection of a very large number of *code* corruptions in a popular Linux distribution [20] and a group membership service [7]. Both studies report that randomly corrupted instructions may corrupt part of the process state or divert the control-flow of the process. Gu *et al.* [20] report that "*detailed tracing of crash dumps indicates that random error injections can corrupt several instructions in a sequence*". This is because the processor (an Intel CPU) may interpret the instruction stream as a sequence of random instructions if it starts reading it from an incorrect location. The same can occur in case of a faulty jump to a location that is not the beginning of a valid instruction. However, these faults are not dangerous, as discussed by the authors: "*As a result, the system executes an invalid sequence of instructions, which is very likely to cause quick (i.e., short latency) crash*". The paper also includes examples of the injected code corruptions, in which a crash always occurs immediately after the corrupted instructions are executed due to invalid arguments or references. The cases of error propagation across processes discussed by Basile *et al.* [7] are also caused by direct state corruptions or by single corrupted instructions that corrupt the state.

Observation 2: Control-flow faults are dangerous. According to the aforementioned fault injection experiments on the Linux code, code corruptions inverting the outcome of branch operations are the most most likely to generate incorrect outputs [20]. The danger of control-flow faults has long been known in highly dependable systems and has lead to the development of a range of techniques for control-flow checking (see for example [32]). Therefore, such faults need to be considered as possible sources of non-crash process faults.

Observation 3: Reported failures are due to transient faults. The relevant failures reported in Table 1 do not seem to be caused by permanent faults being repeatedly activated. Indeed, many faults of Table 1 are explicitly categorized as soft, or transient, errors, something that was already observed by Gray [19]. Hardware-level fault injection campaigns indicate that permanent process faults are likely to manifest as quick crashes [29]. Some memory modules have scrubbing facilities for detecting potential permanent faults proactively [38]. If detected but uncorrectable memory errors in DRAM modules are not discarded by the operating system or application, they might cause non-crash process faults [30]. However, it is common practice to shut down the server and replace the faulty memory module in these cases [38]. Disk subsystems commonly employ specific techniques to recover from some permanent faults such as bad sectors.

3 The Arbitrary State Corruption model

We now introduce our Arbitrary State Corruption (ASC) model, which formalizes realistic faults based on the three observations we made in the previous section. First, the fault model should represent state corruptions, including those caused by error propagation. Second, it should also model control-flow faults leading to incorrect jumps to some correct operations of the same faulty process. Third, it should consider transient errors that could sporadically appear over time, without assuming that the root cause of the problem is diagnosed. The ASC model includes crashes, which represent a trivial case for hardening. We will focus our discussion on faults that cannot be modeled as simple crashes.

The corruption of even a single variable can propagate to multiple other variables, depending on the way the process state is organized. A similar argument arises when considering control flow faults, whose analysis may require deriving complex application-specific control-flow graphs based on the low-level execution blocks of the code [32].

While sophisticated analyses can be conducted if the low-level details of the distributed system implementation are known, we want our hardening technique to be applicable to generic distributed systems with minimal knowledge. We only assume that processes communicate via unreliable message passing and hold a local, initially correct state. Every time a process receives a message, it calls the corresponding event handler, which can modify the local state and produce one or more output messages. We also want the hardening algorithm to be independent of its low-level implementation.

ASC fault model. The ASC fault model simplifies the matter by taking a conservative approach. It considers all data that is locally accessed by a process as its state, and it assumes that a fault may arbitrarily change the values of any number of variables. Faults can also make the control flow jump to any instruction of the process.

Formally, the model abstracts each process π of a distributed system as a set of guarded commands of the form $\langle G(m_{in}), B(S, m_{in}) \rangle$, where S is the process state, m_{in} an input message, $G(m_{in})$ is a boolean activation condition and $B(S, m_{in})$ is an event handler. Event handlers modify the state and produce a list of output messages. Processes are deterministic: any input message activates at most one event handler.

Definition 1 (ASC fault) *An* ASC fault *occurring at a process π either causes π to crash or assigns an arbitrary value to any variable of its process state S, possibly modifying the control flow of π and causing it to execute from an arbitrary process instruction.*

ASC faults can occur multiple times and at any time, but not arbitrarily often: at most one fault can occur during the execution of a single event handler. In our use cases, event handling occurs in the order of a few milliseconds at most, a conservative bound given the fault frequencies reported in the literature (e.g. [38]).

Data integrity. The model assumes a data integrity property that corresponds, in the ASC model, to the cryptographic assumptions limiting the strength of an adversary in the Byzantine fault model.

Data integrity is protected by replicating each variable of S with a variable of a *replica state R*. The following (slightly simplified) definition refers to corruptions of variables in S; the same property also holds if variables in R are corrupted.

Definition 2 (Fault diversity) *Immediately after a fault modifies the value of a variable v of a state S, the value of v in S is different from the value of v in the replica state R until either (i) v is modified by an assignment, or (ii) the replica of v is assigned to a new value obtained by reading v.*

Fault diversity makes it possible to detect data corruptions by comparing replica variables; however, it does not mandate that the two variables will remain different forever, making detection challenging. The above definition admits two cases in which a corrupted variable v takes the same value as its replica. For example, assume that the current correct value of v and its replica is 0, but an ASC fault assigns v the erroneous value 5. The two replicas are different immediately after the fault. The event handler in execution may use the incorrect value of v to determine the value of other variables, and then re-initialize v to 0, making the corruption undetectable again. Such execution is admitted by our definition of fault diversity. If the comparison is executed after v is re-initialized, the corruption is not detected and error propagation can occur, so the execution time of this check is key. Hardening must minimize the number of checks while providing error isolation guarantees.

4 ASC hardening

This section presents our ASC hardening technique. An overview of the steps executed by a fully ASC-hardened process is given in Algorithm 1; a more formal and detailed pseudocode can be found in the accompanying technical report [13]. Instead of directly discussing Algorithm 1, we introduce checks incrementally starting from a simple protocol example. We discuss failure scenarios to guide the derivation of our ASC-hardening technique. Due to space limitation, we cannot exhaustively consider all fault scenarios; we refer to our technical report for a complete correctness proof [13].

We target systems where processes communicate by sending message through unreliable channels, which can duplicate messages, drop them, and send them to the wrong recipients. Message corruptions respect fault diversity if message fields are replicated.

ASC hardening guarantees the following property.

Property 1 (Error isolation) *Let m be a message sent by a faulty hardened process, and assume that some of the variables whose value is included in m have an incorrect value when the message is sent. If a correct hardened process receives m, then it discards m without modifying its local state. If a faulty hardened process receives m and modifies its local state according to m, then it crashes before sending any output message.*

The correct value of a variable is defined inductively for every received message *m*: either *m* is dropped and its receipt has no local effect, or *m* is processed by updating all required variables without faults. Error isolation prevents both *direct* error propagation, from faulty processes to correct processes, and *indirect* error propagation, caused by faulty processes that send incorrect messages in response to another incorrect message.

Error isolation guarantees that, after a fault, a process with a corrupted state will only expose a benign behavior. In many cases the process eventually crashes as a consequence of executing some internal check; however, a fault may corrupt values at any time, even in the "last mile" before the message is actually sent. These faults are harmless because *all* incorrect messages a faulty process will send after a state corruption will be discarded by their recipients.

At a high level, hardening guarantees error isolation by ensuring that, when an output message is sent, the value of any corrupted variable in *S* does not match with the value of its replica in *R*. A sender forms the content of the messages using variables in *S*, and its verification code using the corresponding replicas in *R*. This use of message verification enables the receiver to discard not only messages corrupted by the network, but also messages built using corrupted sender state.

Hardening is not trusted: ASC faults can corrupt any hardening variable introduced in Algorithm 1, as for example the replica state *R*, and let the control flow start from any instruction of the hardened process code.

For illustration purposes, we consider the example of a client of a distributed data store. Algorithm 2 illustrates one event handler of the client. It reads a value *r* from the store, modifies a state variable *v*, and writes back *v*.

Simple checks. There are two straightforward ways to harden a protocol: *message integrity* and *state integrity* checks. Message integrity checks use message codes (e.g., CRC codes) against network corruptions and the boolean activation condition to verify that the correct event handler is being executed. Every time a message is received containing, for example, a value read from the store, it is verified using the message code and discarded if needed. State integrity checks replicate the process state either using a full copy or using codes. Let us call *S* the original state and *R* the replica state. Every time a variable is read from *S*, these checks compare its value with the one stored in *R*. If a mismatch occurs, a process can preserve integrity in this case by crashing.

The limit of these checks is that they assume that a newly updated variable can only get corrupted after being replicated. Consider for example a fault corrupting the variable *new* during the execution of the event handler. Even if *new* is replicated, there is no redundant information protecting this newly computed value yet. The incorrect value propagates both internally, by overwriting both *S* and *R*, and externally to the data store. The simple checks are not sufficient to prevent this problem.

Hardening against faulty computation. Our ASC-hardening technique addresses the previous problem by executing the event handler twice. The first execution accesses *S*, the second *R*. This prevents error propagation of incorrect values between the two states. If the state prior to receiving the READ-REP message of Algorithm 2 is correct, at most one of the two executions may be impacted by a fault, so one of the two states will be correct.

The client builds the WRITE output message using the value of *v* in *S* and its message code using the replica of *v* in *R*. The receiver of a message, the store process in the example, detects incorrect values contained in the message by verifying its code.

Checking the checkers. Our hardening technique does not assume the existence of trusted components such as trusted voters. Checks can also fail, as we discuss with the following example. We have previously assumed that the state before receiving the READ-REP message is correct. Consider now the alternative case where a previous fault had already corrupted *v* and its replica, resulting in a latent error. For example, assume that *v* and its replica had a correct value 0, but previous faults gave them the incorrect values 7 and 8, respectively. When the message is received, the state integrity check typically detects the corruption. However, a fault might invert or skip the outcome of the check, making the value of *v* appear as correct and letting both executions of the event handler produce the same incorrect output.

Our ASC-hardening addresses this problem by checking state integrity during both executions of the event handler. If at least one execution of the event handler is correct, it either computes correct new values using correct inputs or crashes. Instead of modifying *S* directly, the first execution of the event handler now uses copy-

Algorithm 1: Overview of a hardened event handler.

```
1  Message integrity check;
2  First execution of the event handler, using state $S$ –
   during the execution, check state integrity of
   variables the first time they are read, store
   modifications to $S$ in a copy-on-write buffer $N$, and
   store the identifiers of the modified variables in $U$;
3  First at-least-once gate;
4  First at-most-once gate;
5  Second execution of the event handler, using replica
   state $R$ – during the execution, check state integrity
   of variables the first time they are read, and store the
   identifiers of modified variables in $U'$;
6  Second at-most-once gate;
7  Second at-least-once gate;
8  For each variable with identifier in $U$, apply changes
   from $N$ to $S$ and check that the identifier is in $U'$,
   crashing otherwise;
9  Check that $U = U'$ and crash otherwise;
10 First and second at-least-once checks;
11 Message integrity check;
```

Algorithm 2: Example event handler pre-hardening.

```
12 upon receive ⟨READ-REP, $r$⟩ from the store process
   // $v$, $new$ are state variables
13     if $v > 5$ then    $new \leftarrow r + v + 5$;
14     else               $new \leftarrow r + v$;
15     $v \leftarrow new$;
16     send ⟨WRITE, $v$⟩ to the store;
```

on-write access to S, storing incremental changes in a buffer N. In the example, a copy of v is added to N to store its new value. The second execution of the event handler can thus access the original value of S for its state integrity check. The incremental changes of N are made persistent in S after the second execution has completed. The sets of variable identifiers U and U' are used to prevent incorrect state updates in line 8. Our ASC-hardening also verifies message integrity twice.

Handling control-flow faults. We now discuss how to guarantee correctness in spite of control-flow faults. A fault might result in the incomplete execution of an event handler: for example, after updating the value of the variable *new*, the client might not update v, sending the old value of v to the store. Executing the handler multiple times is also incorrect. For example, the client might add r to v twice and send this incorrect value to the store.

Algorithm 1 handles these faults using what we call control-flow gates and checks. For each control flow gate, hardening introduces a different pair of control-flow variables, c and its replica c', and uses a distinct label L. Initially, c and c' are set to a value different from L. The fact that a control flow variable is set to L marks the fact that the process has reached a given point in the execution flow of the hardened event handler. Control flow variables are replicated to prevent faults from incorrectly setting both of them to L. For simplicity, in the following discussion we use the same names for all control flow variables and labels.

An *at-most-once* gate $\mathscr{G}$ is a sequence of three high-level instructions. First, if $\mathscr{G}.c \neq \mathscr{G}.c'$ or $\mathscr{G}.c = L$ then the process crashes. Else, $\mathscr{G}.c$ is set to L and $\mathscr{G}.c'$ is set to L. Note that the full gate cannot be executed twice, even in presence of a fault. These gates are used to prevent situations where instructions of an event handler are executed twice, for example adding r to v twice. The hardened process uses two separate instances of at-most-once gates, at lines 4 and 6 of Algorithm 1.

An *at-least-once* gate $\mathscr{G}'$ is a sequence of two assignments of $\mathscr{G}'.c$ and $\mathscr{G}'.c'$ to L, respectively. An at-least once check verifies that the corresponding gate has been reached by checking that $\mathscr{G}'.c = \mathscr{G}'.c' = L$ and crashing otherwise. There are two separated instances of at-least-once gates in Algorithm 1, at lines 3 and 7. The checks for both gates are at line 10.

We have discussed that the following condition must hold when output messages are sent after the hardened event handler has completed its execution: if the value of a variable in S is incorrect, then it does not match with its replica in R. For variables in S or R whose current value has been last modified by a fault, the condition holds by definition of fault diversity. Therefore, we focus on showing that the condition holds for variables determined by instructions; in this case, fault diversity might not be sufficient.

Consider the case where some variable of S is modified by an instruction. Variables in S are modified only in line 8. Let t_8 be the last time when an instruction of line 8 is executed. If no fault occurs before t_8, then both executions of the even handler in lines 2 and 5 complete correctly before t_8. Variables could be corrupted by executing instructions of the event handler after t_8 if a fault occurs. However, one of the at-most-one gates would lead to a crash.

If no fault occurs after t_8, the at-least once checks of line 10 either crashes the process or guarantees that lines 3 and 7 are executed before t_8. Let t_3 and t_7 be the last time when the two gates are executed, respectively.

If no fault occurs after t_3 or before t_7, then the second execution of the event handler is executed correctly once. The at-most-once gates ensure that this second execution is not repeated. After the second execution is completed, the updates of all variables of R are correct, and the correct identifiers of these updated variables are in U'. Any variable of S that is supposed to be updated but has an incorrect value can thus be detected as faulty by comparing it with its correctly updated replica in R. The checks on U and U' of line 9 ensure that only the right variables of S are updated.

The last case is the one where, if no fault occurs before t_3 or after t_7, the first execution of the event handler updates N and U correctly. By definition of t_3, no fault can occur after t_3 and lead to executing instructions of line 2 again. After t_7, the state S is updated in line 8. If some value of N is corrupted by a fault after being correctly updated, this can be detected due to fault diversity. The checks on U and U' of lines 8 and 9 guarantee that the right updates are executed.

5 The PASC library and use cases

The PASC library automates the hardening of distributed protocols. It is a runtime execution environment inside which user-defined protocols are executed. The user is required to provide PASC with a specification of the *protocol state S*, the *event handlers*, and the *messages* used by the protocol. Each of these components corresponds to an interface that must be implemented by user-defined classes. The library is currently implemented in Java.[1]

The implementation of message-passing primitives is external to hardened processes, and thus to PASC, so the user has no restriction of how to implement them. During initialization, the user must create a new PASC runtime object and pass it references to all the user-defined protocol classes. The protocol is actually run by passing every received message to a method of the runtime, which selects the correct event handler and returns output messages. The runtime takes care of transparently running the hardened event handler.

Particular care is needed when defining the protocol state, a class whose fields are the state variables. PASC transparently builds a replica R of the protocol state S. PASC needs to track accesses to variables in order to implement copy-on-write updates, execute state integrity checks, and build the sets U and U'. We use the names *get-v* and *set-v* for the methods accessing the variable v, where v is a unique label identifying the variable. A "variable" can be any subset of the protocol state. During initialization, PASC encapsulates the state class into a dynamically generated class that intercepts all calls to getters and setters. PASC requires that event handlers use only getters and setters of the state class for reading from and writing to state variables. Checking variables before reading them is a major source of performance overhead, which grows with the size of the variables. The user can reduce this overhead by defining many fine-grained getters instead of few coarse-grained ones. For example, a getter returning an array is more expensive to call than one returning only a single element of the array.

Messages are transparently replicated and verified, but the user needs to specify how the message replica is stored in the message, and how to verify messages. By default, replica messages are CRC32 codes. We used TCP for message passing.

Use case: Paxos state machine replication. We used PASC to implement an ASC-tolerant version of the Paxos SMR protocol, a common fault-tolerant protocol implemented by many practical system (e.g. [10]).

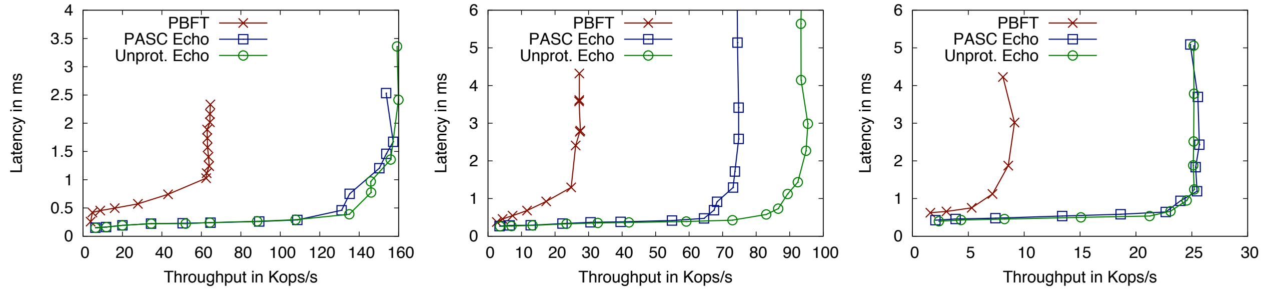
The Paxos protocol uses a leader to propose an execution order of requests. It tolerates the presence of multiple concurrent leaders, but only ensures progress in normal periods with only one leader. The critical path of requests in normal periods is as follows. The leader collects and orders requests from the clients, sending a sequence of operations to all other processes. If a process accepts the order proposed by the leader, it sends an acknowledgement to all other processes. A process executes requests in the sequence indicated by the leader only after receiving acknowledgements from a majority of processes. Clients deliver the first reply they receive.

Paxos tolerates crashes but not data corruptions. The corruption of a single variable of a single process can lead to the unrecoverable loss of the history of executed operations. For example, if the current ballot number a replica stores gets corrupted, then the recovery of a new leader may complete in a state that is inconsistent with the previously committed state. The new leader can then incorrectly overwrite committed state.

In the ASC-tolerant version of Paxos, both client and replica processes are hardened with PASC. The same number of replicas as for crash-tolerance is sufficient: at least $2f+1$ to tolerate f faulty replicas. Like in Zookeeper, the replication protocol itself detects if the current leader is faulty and uses a leader selection algorithm only to elect a new leader.

PASC Paxos guarantees error isolation for the messages exchanged by the consensus layer. This ensures that correct processes execute operations in a consistent order, and that faulty replicas do not process incorrect messages before sending apparently correct messages. Achieving full ASC tolerance, however, requires also handling corruptions to the state of the state machine, not only of consensus. Handling faulty state machines boils down to two simple changes of PASC Paxos compared to the original Paxos. Clients cannot deliver the first reply they receive from any replica; instead, they must wait for a quorum of $f+1$ consistent replies to make sure that at least one reply comes from a correct process. Given the presence of $2f+1$ replicas, a quorum of correct replicas is available. Furthermore, processes periodically take checkpoints of the state machine to enable garbage col-

[1] The library is open source, see https://github.com/yahoo/pasc



(a) Request and reply size: 0kB (b) Request and reply size: 1kB (c) Request and reply size: 4kB

Figure 1: Micro-benchmarks of hardening: Latency-throughput curves.

lection of the operation log. These checkpoints can be undetectably corrupted, so they need to be validated by other processes. This is done by simply exchanging a digest of the state and waiting for $f+1$ confirmations of the digest. There are known techniques to efficiently compute incremental digests of states, as for example the Merkle trees proposed in [9]. With these changes, the replicated state machine does not need to run inside the PASC runtime.

Use case: Eventually consistent storage. ASC hardening can be used as a building block to implement ASC-tolerant state machine replication, but it is not restricted to it. Our second use case shows that it is possible to harden fault-tolerant algorithms with different consistency guarantees.

We designed a simple key-value store, called SimpleKV, which replicates data with relaxed consistency to scale throughput and latency. It implements a single-writer multiple-readers register, a fundamental abstraction in distributed computing. Clients have write access to their own key and read access to all keys, which are replicated by multiple distributed data stores. Each replica holds a copy of all keys. Clients write requests to $f+1$ stores for persistence before returning, where f is the number of faulty stores that must be tolerated. SimpleKV does not need to write to a majority quorum of stores; this ensures scalability of write operations. Clients add a client-local timestamp to write requests. A store overwrites the local value of a key only if it receives a write with a higher timestamp than any other write previously observed for that key. Read operations are served by a single store, which ensures scalability.

SimpleKV guarantees eventual consistency [45]: in periods when no client invokes write operations, all data stores eventually hold the same latest value for all keys. Whenever a data store applies a write to its local state, it forwards the new value in the background to the other data stores, following the process ID order.

6 Evaluation

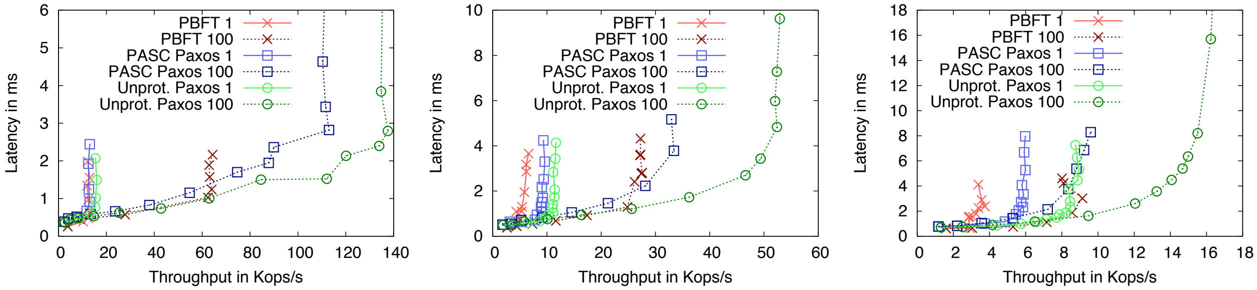
6.1 Performance of PASC

The ASC-tolerant use case protocols show the flexibility of our hardening approach. However, ASC-hardening must also be feasible from a performance viewpoint. In this section, we show that this is the case. We compare the performance of our use cases with and without ASC hardening. PASC has an option to turn off hardening, running just the original processes.

Our evaluation setup consists of servers equipped with a quad-core 2.5 GHz CPU, 16 GB of RAM, and a Linux with kernel 2.6.18. We use a Gigabit network. All protocols are configured to tolerate one fault, so $f=1$. Protocol clients are run on dedicated machines, different from the ones used for replicas and data stores. Each client sends a new request as soon as its previous request is completed. All measurements are started after the clients have issued a few thousands of requests and the system has reached steady-state performance. We disable multicast for all protocols because the network does not support it. All plots show averages over five runs; we observed negligible variance. We control throughput by increasing or reducing the number of clients.

Hardening micro-benchmarks. ASC hardening guarantees error isolation without using state machine replication (SMR). This simplifies the hardening of many practical distributed systems, where processes are loosely coupled. In our eventually consistent SimpleKV store, for example, ASC hardening is sufficient to prevent the propagation of incorrect data and messages; SMR is not needed for fault tolerance.

Hardening with Byzantine faults is much more expensive. Existing work hardens scalable crash-tolerant systems against f Byzantine faults using variants of SMR and replicating each process (*host*) on $3f+1$ different servers (*guards*) [23]. This leads to higher replication costs and increases complexity.



(a) Request and reply size: 0kB

(b) Request and reply size: 1kB

(c) Request and reply size: 4kB

Figure 2: Micro-benchmarks of state machine replication: Latency-throughput curves.

Figure 1 compares the performance of hardening a single process. We use micro-bechmarks, where the hardened process receives a request and sends a reply, without making any computation. We consider request and reply sizes of 0, 1 and 4 kilobytes. For the Byzantine model, we provide the performance of Castro's implementation of the PBFT protocol [9] as a reference value since no implementation of the SMR protocol of [23] is publicly available. Networking is the main bottleneck in the 0 kB and 4 kB cases due to high number of messages and message size, respectively. The overhead of PASC is negligible in these cases. The cost of message verification is more evident in the 1 kB case. In all cases, executing a full BFT SMR protocol is substantially more expensive: PASC improves throughput by about 2.5x.

SMR micro-benchmarks. For state machine replication, we compare our ASC-hardened implementation of the Paxos protocol and Castro's implementation of the PBFT protocol, which can be seen as the enhancement of Paxos for the Byzantine fault model: it also uses a leader to order requests and only executes requests when they have been agreed upon (at least tentatively). Paxos requires $2f+1$ replicas to tolerate f faults, regardless of hardening; PBFT needs $3f+1$ replicas, although it can be extended to use $3f+1$ replicas for agreement and $2f+1$ for execution [47].

We stress the system by considering all-writes workloads where the system must agree on each request. All protocol implementations use batching with congestion window, a technique introduced by PBFT. The leader batches incoming requests as long as agreement on a previous batch of requests is ongoing. We extended this mechanism by letting the leader start agreement on a batch anyway if the number of requests in the current batch reaches a certain threshold. We show results for maximum batch sizes 1 (no batching) and 100 since larger batch sizes did not significantly improve performance. We execute the same micro-benchmarks as the previous experiments. For reference, a typical average request size for ZooKeeper is around 1 kB [25].

The latency-throughput curves of the different protocols are reported in Figure 2. Using batching results in a net throughput improvement and it does not impact latency significantly. Adding CRCs to messages is common in crash-tolerant protocols, so we evaluated its impact on the throughput of Paxos and found that it is negligible. PASC Paxos outperforms PBFT in maximum throughput between 70% and 10%, depending on the size of the message. Paxos outperforms PASC Paxos significantly only with larger messages: message verification becomes more costly, storing messages requires transferring more data due to our use of copy-on-write, and larger areas of memory need to be checked.

The minimal latency when a single client submits requests sequentially is reported in Figure 3. The latency of the three protocols is similar and very low. This is because PBFT uses tentative executions to send replies with the same analytical latency as Paxos [9]. Unlike PBFT, our Paxos implementation processes messages using a pipeline of different stages, which results in slightly higher latency.

Figure 4 reports the steady-state JVM user memory usage. The occupation grows with the loads of the system so we consider runs with peak throughput. Both in case of Paxos and PASC Paxos, the occupation grows rapidly between 0 kB and 1 kB, and remains almost flat for 4 kB. While the replication of the protocol state does result in higher memory occupation, the absolute overhead is small. The relative difference between the two protocols remains around 43-58% regardless of the message size. The state machine is not encapsulated inside PASC and its state is not replicated, so the memory overhead would not grow if we would consider running a state machine on top of PASC Paxos.

ZooKeeper. Micro-benchmarks are good for stress-testing state machine replication protocols. However, these protocols are never run stand-alone. For a more realistic assessment of the performance cost of these sys-

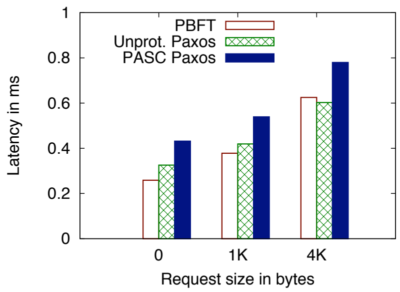


Figure 3: Single request latency

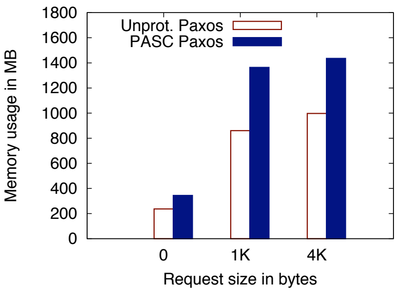


Figure 4: Memory occupation

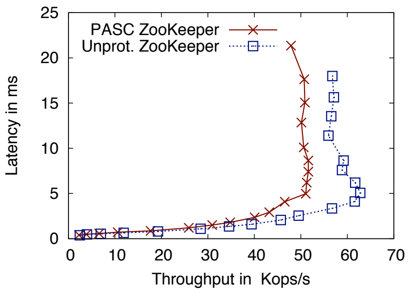


Figure 5: ZooKeeper benchmark

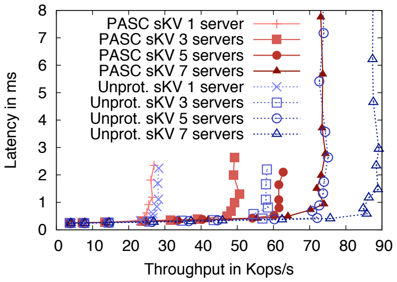


Figure 6: SimpleKV performance and scalability.

tems, we implemented a subset of the ZooKeeper API on top of Paxos. ZooKeeper exposes a tree-like data structure that is stored in memory. We chose the subset of commands that can be implemented using vanilla state machine replication: *create*, *delete*, *exists*, *get data*, *set data*, *get children*. Handling ephemeral nodes and sessions would have required handling timeouts, which are non-deterministic. Watchers require clients to handle replies while they may have other pending operations; this does not directly fit into the concept of well-formed run. For our implementation we use the actual data structure used internally by ZooKeeper to store its state, which is called DataTree, and build glue code to make it communicate with Paxos.

In order to test the system we created a custom ZooKeeper benchmark. Clients issue requests choosing a random command. Each command operates on a random node of a data tree of depth up to 5. Each internal node of the tree has up to 5 children. Figure 5 compares the performance of our ZooKeeper implementation running on Paxos and PASC Paxos. Both in terms of latency and throughput, the overhead of using PASC is less significant than in micro-benchmarks. The peak difference in maximum throughput between the two versions is around 17%. Note that the ZooKeeper state is not replicated in PASC Paxos, since ZooKeeper is a state machine.

SimpleKV. We run SimpleKV with an increasing number of data stores to study the scalability of the protocol with and without PASC. We use a 20/80 benchmark, where clients continuously issue operations that are write in the 20% of the cases and reads in the remaining 80%. The size of requests and replies is 1 kB. Figure 6 shows the latency-throughput curve resulting from our experiments. Both the original SimpleKV protocol and PASC SimpleKV scale almost linearly with the number of servers. A BFT-hardened version of SimpleKV would have used most of the processes for replication, using SMR groups, rather than for better performance.

6.2 Coverage of the ASC model

We now study how random faults affect the operation of our Paxos implementation; the results support our claim that the ASC model is a proper approach to capture the key properties of these faults. We follow closely the established approach of works like Gu *et al.* [20] and Basile *et al.* [7]: injecting single bit flips. We increase the likelihood of activating the injected faults by using stack traces or targeting *core classes*, which are selected by profiling the execution of Paxos. The core classes are the top ten CPU- and memory-intensive classes (considering all their instances) in each of the following categories: Paxos classes, PASC-runtime classes, Java library classes, messaging classes (we used Netty), and JVM-internal classes.

We inject faults at the Paxos leader because it is the process with the highest likelihood of propagating errors. After injecting faults, we wait approximately one minute for its manifestation before interrupting the experiment. We report results of our injections during the normal execution of the protocol, where a single leader is present in the system. Some fault injections result in a leader crash and trigger a subsequent recovery. We also reproduced worst-case scenarios where the leader crashes and some other replica has a corrupted state, and found no case of error propagation using PASC.

We consider four types of injections, corrupting *bytecode*, *binary code*, *pointers* and *primitive values*.

Bytecode injection: In each run, we select a random

	Code corruptions				State corruptions			
	bytecode		binary code		pointer		value	
	unprot.	hardened	unprot.	hardened	unprot.	hardened	unprot.	hardened
Undetected error propagation	0	0	3	0	66	0	27	0
Detection through hardening	-	0	-	1	-	164	-	166
Crash or hang	956	1028	684	635	2165	2024	136	42
Total number of runs	1818	1818	1047	1047	4000	4000	1237	1237

Table 2: Results of fault injection experiments

method among core classes and flip one random bit on its bytecode before this is loaded by the JVM. We disable JVM bytecode verification at loading time.

Binary injection: At a random time during the execution of the protocol, we take the stack trace. We then inject bit flips around 1 to 10 random return addresses selected among those found in the stack, at a random location within a range of 400 bytes.

Pointer and value injection: At a random time during the execution of the protocol, we corrupt one random field of a random instance of a random core class. We restrict to non-primitive values for pointer injections and to primitive values for value injections.

Results. Table 2 summarizes the results of our fault injection experiments. We partition runs into three classes, based on the effect of the injected fault: runs where *undetected error propagation* occurs; runs with no error propagation where an internal error of the faulty process or an incorrect message is *detected through hardening*; runs where no error is propagated and no internal error or incorrect message is detected through hardening, but the faulty process *crashes or hangs* anyway; and runs where the injected fault has no effect (not shown).

Our main conclusion is that hardening is both necessary and sufficient to prevent undetected error propagations in our experiments: error propagation does occur, but only without hardening. Most code injections result in crashes if the error is activated. A few cases of potential error propagation due to code injection occur if processes are not protected, but ASC-hardening is sufficient to guarantee error isolation. State corruptions in pointers and values are much more likely to generate error propagation. These findings support our choice of modeling all random data corruptions as ASC faults.

7 Related work

The rigorous design and proof of correctness of fault-tolerant distributed algorithms require the formalization of a system model. In the selection of such a model there is always a tension between several facets: how well it describes reality, the classes of problems it allows solving, and the efficiency of the algorithms that assume it [37]. Our ASC fault model represents a new tradeoff for practical fault-tolerant systems. It was inspired by Dijkstra's seminal work on self-stabilization, where the whole system state can transition to arbitrary values [15]; however, the goal of ASC hardening is isolating faulty processes rather than converging to a correct global state.

Some theoretical work proposed transformations to improve the fault tolerance of distributed algorithms from crashes to Byzantine faults, mainly targeting crash-tolerant agreement algorithms where processes communicate by broadcasting information over multiple communication rounds [12], or synchronous systems [35]. NewTOP makes strong synchrony assumptions and may not tolerate the failures of two replicas [34]. Baldoni *et al.* use failure detection modules tailored to a round-based consensus protocol in order to detect some faults beyond crashes [6]. ASC hardening only makes very basic assumptions about the structure of distributed processes; it is not restricted to agreement protocols, it does not depend on synchrony assumptions, and it does not require designing protocol-specific checks. Nysiad is a more generic hardening technique that does not require synchrony for progress; as discussed in our evaluation, it hardens against Byzantine faults, but it uses a variant of state machine replication as building block [23]. Byzantine fault detection is a viable alternative to detect integrity violations after they have occurred, but it does not guarantee error isolation [21].

Hypervisor-based approaches run multiple replicas of a process inside the same physical machine. Traditionally, these approaches targeted crash or fail-stop models that do not encompass ASC faults [8]. Furthermore, unlike ASC hardening, they assume failure-independence of local process replicas and a trusted hypervisor.

Techniques for error detection come at least from the 50s with Hamming's work. Taylor et al. proposed adding redundant data to data structures to detect corruptions of their instances [41]. Detection of control-flow corruptions in software has also a long tradition. Tront et al. proposed a simple technique to detect control-flow corruptions that force software to jump to a different subroutine [42]. The technique uses a tag or signature to check in which routine processing should be at the moment. Several variations of this idea have later appeared.

ASC hardening uses knowledge about the structure of distributed programs to guarantee end-to-end error isolation properties and execute very few checks. There exist many other hardening techniques in the literature that do not assume knowledge of the hardened program. For generality, they trade higher performance overhead, lack of end-to-end guarantees, or a less comprehensive fault model. For example, SWIFT is a compiler-based technique that targets a single event upset, where a single bit is flipped once [36]. Unlike our hardening technique, SWIFT assumes that all memory errors are detected by hardware error correcting codes. SWIFT executes twice every binary instruction computing new values, and compares the two obtained values immediately afterwards; this potentially results in high overhead, although a performance comparison with unprotected code is not given in [36]. ASC hardening executes full event handlers twice too, but it does not execute comparisons after every operation. Unlike ASC, SWIFT does not protect against faults corrupting values after the comparison and before storing them in memory, so it does not fully guarantee error isolation. Finally, SWIFT's control flow signatures detect a subset of the control flow errors covered by our technique.

An older low-level hardening technique is AN-coding, where instructions generate encoded variables from encoded variables [16]. AN-codes protect the algorithm against some subsets of hardware faults (e.g. operator corruptions, but not exchanged operators). Software Encoded Processes (SEP), proposed in [46], run on top of an encoded interpreter, which uses AN-codes and other coding techniques. The fault coverage of SEP is probabilistic and depends of the number and distribution of the bit flips. Being low-level, SEP leads to high overhead, with slowdowns between 2.2x and 25x.

Yoo *et al.* use local checkpoints to harden protocols against crashes [48]. ASC-hardening is orthogonal: it contains error propagation with ASC faults but requires explicit handling of crashes and message omissions.

8 Conclusion

We proposed ASC-hardening as a novel, sound approach to systematically protect the integrity of distributed systems against realistic faults. Hardening prevents error propagation across processes by converting state corruptions into process crashes and message omissions. We have implemented a library, PASC, to transparently harden processes and evaluated it using a few realistic examples: state-machine replication, a key-value store implementation, and a subset of the ZooKeeper coordination service. Our performance evaluation showed that PASC enables excellent performance while inducing an acceptable penalty; for our ZooKeeper benchmark, we obtained over 50k ops/s and roughly a 17% throughput drop compared to the unprotected version. Our fault injection experiments also showed that PASC is able to effectively prevent error propagation upon data corruptions. Preventing error propagation is critical to avoid massive outages in many production systems.

Acknowledgements

We would like to thank our shepherd Andreas Haeberlen, Jon Howell, and the anonymous reviewers for the valuable feedback. This work has been partially supported by the SRT-15 project (ICT-257843), funded by the European Community. Marco Serafini and Flavio P. Junqueira also acknowledge support from the INNCORPORA - Torres Quevedo Program from the Spanish Ministry of Science and Innovation, co-funded by the European Social Fund. Miguel Correia was partially supported by FCT through the Multi-annual PIDDAC Program funds, project RC-Clouds and scholarship SFRH/BSAB/992/2010.

References

[1] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of USENIX OSDI* (2002), pp. 1–14.

[2] AMAZON. Amazon S3 availability event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`, July 2008.

[3] AMAZON. S3 data corruption? `https://forums.aws.amazon.com/thread.jspa?threadID=22709`, June 2008.

[4] AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., AND LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (Jan-Mar 2004), 11–33.

[5] BAIRAVASUNDARAM, L. N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., GOODSON, G. R., AND SCHROEDER, B. An analysis of data corruption in the storage stack. *ACM Transactions on Storage 4*, 3 (2008), 1–28.

[6] BALDONI, R., HELARY, J.-M., AND RAYNAL, M. From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach. In *Proc. of IEEE DSN* (2000), pp. 273–282.

[7] BASILE, C., LONG, W., KALBARCZYK, Z., AND IYER, R. Group communication protocols under errors. In *Proc. of IEEE SRDS* (2003), pp. 35–44.

[8] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. *ACM Transactions Computer Systems 14*, 1 (February 1996), 80–107.

[9] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems 20*, 4 (2002).

[10] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: An engineering perspective. In *Proc. of ACM PODC* (2007), pp. 398–407.

[11] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. In *Proc. of USENIX OSDI* (2006), pp. 205–218.

[12] Coan, B. A. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Transactions on Computers 37*, 12 (Dec. 1988), 1541–1553.

[13] Correia, M., Ferro, D. G., Junqueira, F., and Serafini, M. Models and algorithms for ASC hardening and a correctness proof. Yl-2011-003, Yahoo! Labs, 2011.

[14] Dell, T. J. A white paper on the benefits of Chipkill - correct ECC for PC server main memory. Tech. rep., IBM Microelectronics Division, 1997.

[15] Dijkstra, E. W. Self-stabilizing systems in spite of distributed control. *Communications of ACM 17* (November 1974), 643–644.

[16] Fiorin, P. Vital coded microprocessor principles and application for various transit systems. In *IFAC/IFIP/IFORS Symposium* (1989), pp. 79–84.

[17] Gashi, I., Popov, P. T., and Strigini, L. Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. *IEEE Transactions on Dependable and Secure Computing 4*, 4 (2007), 280–294.

[18] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proc. of ACM SOSP* (2003), pp. 29–43.

[19] Gray, J. Why do computers stop and what can be done about it? In *Proc. of IEEE SRDS* (1986), pp. 3–12.

[20] Gu, W., Kalbarczyk, Z., Iyer, R. K., and Yang, Z. Characterization of Linux kernel behavior under errors. In *Proc. of IEEE DSN-DCCS* (2003), pp. 459–468.

[21] Haeberlen, A., Kouznetsov, P., and Druschel, P. Peerreview: Practical accountability for distributed systems. In *Proc. of ACM SOSP* (2007), pp. 175–188.

[22] Hamilton, J. Observations on errors, corrections, and trust of dependent systems. `http://perspectives.mvdirona.com/2012/02/26/ObservationsOnErrorsCorrectionsTrustOfDependentSystems.aspx`, Mar. 2012.

[23] Ho, C., van Renesse, R., Bickford, M., and Dolev, D. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proc. of USENIX NSDI* (2007), pp. 175–188.

[24] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proc. of USENIX ATC* (2010).

[25] Junqueira, F., Reed, B., and Serafini, M. Zab: High-performance broadcast for primary-backup systems. In *Proc. of IEEE DSN* (2011), pp. 245–256.

[26] Knight, J. C., and Leveson, N. G. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering 12*, 1 (1986), 96–109.

[27] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. of ACM SOSP* (2007), pp. 45–58.

[28] Lamport, L. The part-time parliament. *ACM Transactions Computer Systems 16*, 2 (May 1998), 133–169.

[29] Li, M.-L., Ramachandran, P., Sahoo, S. K., Adve, S. V., Adve, V. S., and Zhou, Y. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proc. of ACM ASPLOS* (2008), pp. 265–276.

[30] Li, X., Huang, M. C., Shen, K., and Chu, L. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proc. of USENIX ATC* (2010), pp. 6–16.

[31] Ma.gnolia. Ma.gnolia data recovery status. `http://getsatisfaction.com/magnolia/topics/ma\_gnolia\_data\_recovery\_status`, February 2009.

[32] Mahmood, A., and McCluskey, E. J. Concurrent error detection using watchdog processors-a survey. *IEEE Transaction on Computers 37*, 2 (February 1988), 160–174.

[33] Merideth, M., Iyengar, A., Mikalsen, T., Tai, S., Rouvellou, I., and Narasimhan, P. Thema: Byzantine-fault-tolerant middleware for web-service applications. In *Proc. of SRDS* (2005), pp. 131–140.

[34] Mpoeleng, D., Ezhilchelvan, P., and Speirs, N. From crash tolerance to authenticated Byzantine tolerance: A structured approach, the cost and benefits. *Proc. of IEEE DSN* (2003), 227–236.

[35] Neiger, G., and Toueg, S. Automatically increasing the fault-tolerance of distributed systems. In *Proc. of ACM PODC* (1988), pp. 248–262.

[36] Reis, G., Chang, J., Vachharajani, N., Rangan, R., and August, D. SWIFT: Software implemented fault tolerance. In *Proc. of IEEE/ACM CGO* (2005), pp. 243–254.

[37] Schneider, F. B. What good are models and what models are good? *Distributed systems (2nd Ed.)* (1993), 17–26.

[38] Schroeder, B., Pinheiro, E., and Weber, W.-D. DRAM errors in the wild: A large-scale field study. In *Proc. of ACM SIGMETRICS* (2009), pp. 193–204.

[39] Serafini, M., Bokor, P., Dobre, D., Majuntke, M., and Suri, N. Scrooge: Reducing the costs of fast Byzantine replication in presence of unresponsive replicas. In *Proc. of IEEE DSN* (2010), pp. 353–362.

[40] Song, Y. J., Junqueira, F., and Reed, B. BFT for the skeptics. In *Proc. of BFTW*^{3} (2009).

[41] Taylor, D. J., Morgan, D. E., and Black, J. P. Redundancy in data structures: Improving software fault tolerance. *IEEE Transactions on Software Engineering 6*, 6 (1980), 585–594.

[42] Tront, J. G., Armstrong, J. R., and Oak, J. V. Software techniques for detecting single-event upsets in satellite computers. *IEEE Transactions on Nuclear Science 32*, 6 (Dec. 1985), 4225–4228.

[43] Veronese, G. S., Correia, M., Bessani, A. N., C., L., and Verissimo, P. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*. To appear.

[44] Vijaykrishnan, N. Soft errors: Is the concern for soft-errors overblown? In *Proc. of IEEE ITC* (2005), pp. 2–12.

[45] Vogels, W. Eventually consistent. *Communications of the ACM 52*, 1 (2009), 40–44.

[46] Wappler, U., and Fetzer, C. Software encoded processing: Building dependable systems with commodity hardware. In *Proc. of SAFECOMP* (2007), pp. 356–369.

[47] Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., and Dahlin, M. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. of ACM SOSP* (2003), pp. 253–267.

[48] Yoo, S., Killian, C., Kelly, T., Cho, H. K., and Plite, S. Composable reliability for asynchronous systems: Treating failures as slow processes. In *Proc. of USENIX ATC* (2012).