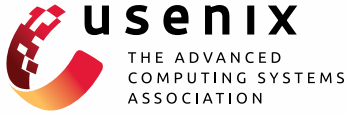

Scalable Error Isolation for Distributed Systems

Diogo Behrens, *Technische Universität Dresden;* Marco Serafini, *Qatar Computing Research Institute;* Sergei Arnautov, *Technische Universität Dresden;* Flavio P. Junqueira, *Microsoft Research Cambridge;* Christof Fetzer, *Technische Universität Dresden*

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/behrens

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Open Access to the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15) is sponsored by USENIX

Scalable error isolation for distributed systems

Diogo Behrens[*], Marco Serafini[◇], Sergei Arnautov[*], Flavio P. Junqueira[‡], Christof Fetzer[*]

[*]Technische Universität Dresden, Germany

[◇]Qatar Computing Research Institute, Qatar

[‡]Microsoft Research, Cambridge, UK

Abstract

In distributed systems, data corruption on a single node can propagate to other nodes in the system and cause severe outages. The probability of data corruption is already non-negligible today in large computer populations (*e.g.*, in large datacenters). The resilience of processors is expected to decline in the near future, making it necessary to devise cost-effective software approaches to deal with data corruption.

In this paper, we present SEI, an algorithm that tolerates Arbitrary State Corruption (ASC) faults and prevents data corruption from propagating across a distributed system. SEI scales in three dimensions: memory, number of processing threads, and development effort. To evaluate development effort, fault coverage, and performance with our library, we hardened two real-world applications: a DNS resolver and `memcached`. Hardening these applications required minimal changes to the existing code base, and the performance overhead is negligible in the case of applications that are not CPU-intensive, such as `memcached`. The memory overhead is negligible independent of the application when using ECC memory. Finally, SEI covers faults effectively: it detected all hardware-injected errors and reduced undetected errors from 44% down to only 0.15% of the software-injected computation errors in our experiments.

1 Introduction

Distributed systems running in modern data centers must be tolerant to faults. Since machine and process crashes are commonly observed, the crash fault model is the one typically adopted. In fact, many systems critical to Web-scale online services are successfully using techniques such as state machine replication [6, 17, 22, 31] to guarantee availability despite crash faults.

The crash model does not cover data corruption faults, which might lead to errors propagating through incorrect messages after a fault. Incidents occurring in large Internet services in the recent past already indicate that data corruption can cause process state corruption [20], data loss [2, 3, 4], or in some unlucky case of error propagation, even multi-hour outages of entire services [1]. This is not surprising: several large scale studies show that faults that would be very unlikely in a small cluster become much more likely at scale and tend to reappear more frequently after the first occurrence [25, 32, 43, 44, 50]. There can be several reasons for data corruption symptoms, for example, manufacturing problems, overheating, an incorrect use of dynamic voltage scaling, hardware/software incompatibility, or power supply faults [2, 36]. For example, we used dynamic voltage scaling while running `memcached` on a single processor with a lower voltage level and found that undetected error propagation occurred in 4 out of 468 runs (see Section 7.2). These problems are in fact known to datacenter operators dealing with large server populations.

New processor generations have traditionally achieved higher performance through higher circuit density and lower energy consumption. This approach, however, has reached physical limits that affect hardware-level reliability negatively [15, 21]. The rate of transient errors for processors has been rising [14, 16] and it might reach up to one user-visible failure per day per chip with 16 nm technology [26, 51].

These trends are already changing the way large-scale distributed systems are designed today. Mesa, a data warehousing system for business-critical data used in Google, uses application-level integrity checks to deal with transient data corruption during computation, which is common at scale [28].

We argue that preventing end-to-end error propagation due to data corruption, including corruption in the computation, is an important requirement for large-scale fault-tolerant distributed systems. Application-specific solutions like the ones used in Mesa leave application developers with the burden of guaranteeing data integrity.

Instead, we advocate for *hardening* approaches that enable crash-tolerant systems to handle data corruption with minimal changes to the application code base.

Existing hardening solutions are not sufficiently cost-effective at scale: they either rely on expensive servers with hardware-level redundancy [7, 53] or require process replication and coordination over multiple physical machines even to achieve simple error isolation [30]. Recent software-based approaches ensure end-to-end error isolation in distributed systems without physical replication, but they still increase the application state by a factor of two and do not support multithreading [12, 23]. The widespread use of multi-core machines with large main memory requires solutions that scale well with many application threads and a large in-memory state.

We introduce Scalable Error Isolation (SEI), a scalable hardening algorithm that transforms the processes of an arbitrary event-based distributed system to ensure *error isolation*. With error isolation, local errors cannot propagate to other processes via output messages in an undetectable manner. SEI is designed to formally guarantee error isolation in the presence of Arbitrary State Corruption (ASC) faults, a well defined and general fault model [23]. We have implemented a C library called `libsei` to harden distributed systems using the SEI algorithm. Hardening is semi-automated: a developer simply annotates, using the `libsei` API, the portions of the code of a distributed system process that are responsible for handling messages, as well as the input and output messages. The compiler then automatically hardens the implementation by instrumenting it with `libsei`.

SEI and `libsei` are scalable in three dimensions. For **memory**, the additional state and redundant information they use is small and independent of the memory usage of the hardened process. For **computation**, they support multithreading and thus cover complex error propagation patterns among threads sharing the same state. For **development effort**, they enable hardening real-world applications with minor developer involvement.

SEI detects *data corruption in the computation*, *i.e.*, errors in the arithmetic and logic units of the processors and in the register files, by processing each message twice and comparing the results locally. The SEI hardening code itself is untrusted, so checks might be incorrect or skipped due to faults. Consequently, SEI runs multiple integrity checks for data and control-flow errors.

SEI detects *data corruption in memory* using compact error detection codes. These codes can be implemented in software, but SEI can also leverage hardware-level mechanisms, such as ECC or Chipkill in DRAM memory modules, to virtually eliminate the memory overhead. These hardware mechanisms both perform this part of the hardening very efficiently and are effective for data in memory [32, 50]. Given that there is no expectation that memory error rates will increase [15, 51], they are likely to remain effective.

To show that hardening existing systems with a small amount of effort is possible, we have hardened real-world applications: `memcached`, a popular in-memory distributed cache system, and Deadwood, a recursive DNS resolver. Hardening these systems required a good understanding of the code base but only small changes.

We conducted extensive fault injection experiments, both software- and hardware-implemented, and a performance evaluation of our hardened applications. We injected faults at hardware level by reducing the CPU voltage and observed that SEI detected all errors. We also injected targeted data corruption faults *during computation* and found that SEI makes the likelihood of error propagation under these faults negligible: from 44% down to only 0.15% of the errors.

Performance results show that the overhead depends on the original bottlenecks of the system. In the case of `memcached`, the application is not CPU-bound so there are spare cycles available for additional processing and the overhead is negligible. Deadwood, however, is CPU intensive so its throughput is reduced to nearly one half.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 introduces the system and fault models. Section 4 presents the SEI algorithm. Sections 5 and 6 discuss the `libsei` implementation and our experience hardening `memcached` and Deadwood. Sections 7 and 8 present our fault injection and performance results. Section 9 concludes this work.

2 Related work

We now discuss the most related approaches for error isolation in distributed systems.

Byzantine fault tolerance. Given the body of work on Byzantine Fault Tolerance (BFT), it is natural to consider Byzantine faults to cover data corruption [18, 37, 40]. The Byzantine model assumes a powerful adversary, and consequently a Byzantine-tolerant system is able to cope with data corruption. Byzantine-tolerant protocols, however, also tolerate faults that are orthogonal to resilience against data corruption, such as intrusions or bugs, as long as replicas use diverse implementations to guarantee fault independence [19]. In particular, intrusions fall into the domain of security, which is often treated as a separate concern in data center applications [13].

Deadwood and `memcached` are instances of a large class of systems in which integrity (safety) is sufficient and continuous availability is not strictly necessary. In the Byzantine model, providing just safety does not significantly reduce cost; see for example Nysiad, which achieves safety through replication and agreement [30].

The Thema system also shows the additional complexity of building Byzantine-tolerant three-tiered Web systems [42]. In contrast to BFT, SEI is lightweight, not requiring replication nor complex agreement protocols. SEI can still be used to harden replicated systems, *e.g.*, replicated state machines based on Paxos [39]. Finally, BFT primarily targets single-threaded state machines. Eve runs multithreaded state machines in multi-core systems by leveraging commutative operations [35]. SEI supports regular multithreaded applications and does not rely on commutativity.

Software-level error detection. Software-level error detection has been subject of a large body of research work. Most such techniques do not provide end-to-end guarantees in distributed systems under ASC faults (see for example SWIFT [48] or [47]). Recent work has proposed using encoded execution to harden distributed systems and provide end-to-end guarantees [12]. Encoding presents important drawbacks, however. First, it induces a significant overhead, showing a response time increased up to 20 times even at modest request loads. For a service such as `memcached` that is sensitive to latency, such an overhead is not acceptable. Second, encoding blows up the application state by a factor of two. Optimizations for both issues exist [10], but are limited to state machine replication. In contrast, SEI presents moderate overhead and has a small memory footprint with hardware error detection codes.

PASC is a hardening algorithm that, like SEI, guarantees coverage of ASC faults, is untrusted, and does not require physical replication [23]. PASC can be used to harden state machine replication: a comparison between BFT and an ASC-tolerant version of Paxos is presented in Correia *et al.* [23]. However, PASC is not scalable in any of the three dimensions indicated by SEI: it does not support processes with multiple threads, it doubles the memory requirements of the hardened application, and it requires implementing the distributed system from scratch using a fixed template. The first two points limit its applicability to multithreaded, memory-intensive systems like `memcached`. The last point makes it hard to harden existing code bases, also because all state accesses must be mediated by a single state object [9].

3 System and fault model

This section presents an informal description of our system and fault model, which is an adaptation of the ASC fault model [23] to multithreaded settings. We refer to our technical report [11] for a complete formalization.

System model. SEI targets event-based processes of distributed systems. Processes consist of one or more threads that spin over three phases:

- *Dispatching* receives a new event (message) and selects an event handler;
- *Handling* executes the actual system logic;
- *Output* sends out messages produced by the event handler.

Threads read from and write to *state variables*, which collectively form the state of the process. These variables persist across the multiple event handling cycles and can be shared among threads. A thread might also have a *local state*, which encompasses the variables that are instantiated every time a handler is executed, but do not persist across handler executions. The state of a process includes all state that is directly observed by its threads and used to determine their behavior. In this work, we consider only state stored in memory, but the model could also be extended to disk storage.

The event handling logic is deterministic, *i.e.*, the state updates and outputs it produces depend uniquely on the input message and the values returned by its reads from the process state. However, we do not require deterministic thread scheduling. Threads can be scheduled in any order and preempted arbitrarily.

Threads can interact through shared variables, which are only accessed in critical sections protected by locks. While this requirement does not cover applications using lock-free state sharing, it represents a very common approach. We assume that threads use lock hierarchies [29], a standard technique to avoid circular waits and deadlocks. Lock hierarchies determine a fixed total order among all locks, and threads acquire and release the locks they need according to this order.

Fault model. We consider a conservative fault model for transient hardware faults. An *Arbitrary State Corruption (ASC) fault* can either crash the process or modify its state by assigning an arbitrary value to any number of its variables. A fault can also corrupt the program counter and make it jump to a different instruction.

This fault model admits worst-case state corruption scenarios. Faults can modify any number of variables and assign them any value. Corruption can occur while data is stored, for example, in main memory, or data is computed, for example, by the combinational logic of a processor. Since it is difficult to determine precisely which part of a process state can be corrupted by a hardware malfunction, a worst-case model is easier to generalize over different applications and platforms.

To guarantee data integrity, the ASC model assumes that it is possible to implement reliable *integrity checks* to detect data corruption while data is stored, for example, in memory, or transmitted as a message. Formally, a variable v in the process state is accessed with the instructions $read(v)$ and $write(v, val)$, where val is the new value of v. The integrity check of a variable v is performed by calling $check(v)$. The model assumes the fol-

lowing *corruption coverage* property of integrity checks: if the current value of a variable v has been determined by an ASC fault, then *check(v)* detects the error by evaluating to FALSE. The ASC model does not specify how integrity checks are implemented. Possible options are Cyclic Redundant Codes (CRC) or hardware techniques like ECC memory. The corruption coverage property is consistent with these techniques: *check(v)* cannot detect errors if the current value *val* of v has been written by a *write*(v, val) instruction. With an ECC implementation of integrity checks, errors that are detected and not corrected correspond to *check(v)* calls returning FALSE right before *read(v)* calls. The *write*(v, val) calls overwrite the memory locations storing v and update their ECCs.

ASC faults cannot occur infinitely often. An execution of the system comprises an unbounded number of thread steps. Such an execution might include an unbounded number of faults, but at most one fault occurs during any event-handling phase of any thread (multiple faults can still occur before that thread starts the next event-handling phase). This assumption does not limit the number of faults in an execution, but it limits *fault frequency*; we justify this assumption as follows. The distributed systems we target handle events such as processing an incoming message in no more than a few milliseconds. The fault model consequently assumes that no two faults happen within such a short time window. The frequency of *uncorrectable* hardware-level data corruption previously reported indicates that this assumption holds with very high probability [32, 43, 50]. It holds even if we consider *hard* errors that occur intermittently at the same memory location [32].

After an ASC fault, computation is expected to continue according to the specification, although perhaps from an instruction that is inconsistent with the previous execution flow and from a wrong state. Transient text-segment corruption typically results in ASC faults: an incorrect operation might corrupt some of the operands, update the wrong variable, or result in an incorrect jump. Related work on the Ensemble system confirms this observation [8]. These faults are often enough tolerated by ASC-hardening, and in fact, the injection of text-segment faults on an ASC-hardened Paxos implementation has shown no case of error propagation [23]. In our hardened `memcached`, text-segment corruption resulted in error propagation with negligible probability (3 out of 7000 errors). There are, however, some cases of text-segment corruption, like some corruptions of load and store instructions, that cannot be tolerated in our model.

4 The SEI hardening algorithm

The SEI algorithm takes an event-based process as specified in the previous section and transforms it into a *hardened process*. A hardened process executes the same application logic as the original process, together with additional checks against ASC faults.

The SEI algorithm comprises a set of transformation rules that introduce both redundant execution and additional verification steps to the original code. To prevent error propagation, SEI might induce a faulty process to abort (*i.e.*, crash) if an internal corruption is detected, or might make a correct process discard a corrupt message it has received. While a faulty process might still send out incorrect messages, incorrect messages do not appear correct to receivers.

In a distributed system where all processes are hardened, SEI guarantees the following properties [11]:

- *Error isolation:* A correct process discards any corrupt input message.
- *Accuracy:* Hardening never causes a correct process to crash or discard a correct input message.

Messages are either correct or corrupt. A correct message is informally defined as follows. Let p be a process and s be the sequence of correct messages p received before sending a message m. Message m is *correct* if and only if there exists a subsequence h_m of s, called a *generation history*, such that the correct behavior of p after receiving only h_m would be to output m. By induction, each generation history of each output message that p produces after sending m extends some generation history of m. In presence of multiple threads processing input messages in parallel, there exists also a consistent interleaving of steps that generates all output messages.

With error isolation, a distributed system designed to tolerate crashes and message omissions is guaranteed to also tolerate ASC faults once its processes are hardened. Accuracy rules out trivial ways to achieve error isolation.

Before discussing details of SEI, we show with an example how to harden an event handler against a small subset of ASC faults by progressively adding checks.

4.1 SEI by example

We illustrate the hardening transformation by presenting an example in which a simple event handler eliminates all but the least significant digit (modulo 10) of the state variable X using a temporary variable V (see Figure 1a). In this example, a fault might corrupt X before the process sends out the message containing X and propagates the incorrect value of X to another process.

A first improvement is to duplicate instructions and variables (see Figure 1b). This approach resembles EDDI [46] and SWIFT [48]. Instruction duplication prevents error propagation if a fault only corrupts *one* variable (*i.e.*, V, V', X, or X') *before* the check of Line 6. Instruction duplication, however, cannot detect several other corruption scenarios. As an example, consider

a) Original code

```
   // handler
1  V ← X
2  V ← V % 10
3  X ← V
   // send message
4  send X
```

b) Instruction duplication

```
   // handler
1  if X ≠ X' then Abort
2  V ← X
3  V' ← X'
4  V ← V % 10
5  V' ← V'% 10
6  if V ≠ V' then Abort
7  X ← V
8  X' ← V'
   // send message
9  send X
```

c) PASC-like

```
   // Execution 1
1  if X ≠ X' then Abort
2  V ← X
3  V ← V % 10
4  N ← V
   // Execution 2
5  if X ≠ X' then Abort
6  V ← X'
7  V ← V % 10
8  X' ← V
   // Validation
9  if N ≠ X' then Abort
   // Final update
10 X ← N
   // send message
11 send X, CRC(X')
```

d) Simplified SEI

```
   // Execution 1
1  V ← X if X = X' else Abort
2  V ← V % 10
3  O ← X if X = X' else Abort
4  X, X' ← V
   // Reset
5  N ← V
6  X, X' ← O
   // Execution 2
7  V ← X if X = X' else Abort
8  V ← V % 10
9  X, X' ← V
10 C ← CRC(V)
   // Validation
11 if N ≠ X then Abort
   // send message
12 send X, C
```

Figure 1: Hardening of a handler that updates a state variable X via a temporary variable V, then sends X out. Primed variables are replica variables (*e.g.*, X' is a replica of X). Comparisons among replicas (*e.g.*, **if** $X = X'$) implement the $check(v)$ operation. "←" represents either $read(v)$ or $write(v, val)$ or a combination of them.

a *last-mile fault*: a fault corrupting V or X between Lines 6 and 9. Even adding a further check comparing X to X' before Line 9 does not help because a fault could still occur after the check.

Another example where instruction duplication falls short is *multi-variable corruption* (recall that a single ASC fault may corrupt any number of variables). By the corruption coverage property, an ASC fault alone cannot corrupt X and make $check(X)$ true, *i.e.*, a fault corrupting X and X' has to result in $X \neq X'$. However, if an ASC fault corrupts X and X' between Lines 1 and 2, then the execution of the subsequent instructions can let V and V' have incorrect but equal values in Line 6. For example, if initially $X = X' = 101$ and an ASC fault between Lines 1 and 2 results in $X = 10$ and $X' = 20$, then $V = V' = 0$ in Line 6. X is corrupt but equal to X' in Line 9, and consequently the error is propagated.

The next step is to apply the checks in the PASC hardening algorithm [23], which we show in Figure 1c. With this approach, we first execute all instructions using the original variables (*i.e.*, X), and then execute all instructions using the replica variables (*i.e.*, X'). During the first execution, the process stores in N the updates to the state variable X (Line 4). In the second execution, it writes directly into X'. Next, the process compares the updates in N and X' and applies them to X. Finally, the process sends a message containing X and its replica X'.

The PASC-like approach detects last-mile corruption scenarios by adding X' to the output message. If a fault corrupts X right before the message is sent in Line 9, the receiver detects it by comparing X and X'. In practice, it is not necessary to send the full value of X' and a CRC is sufficient. Also important, the PASC-like algorithm can detect multi-variable corruptions: the process detects a fault corrupting X and X' between Lines 1 and 2 via the check of Line 5, since X is not modified by Execution 1.

If ECC memory is available, the hardware can perform the checks of Lines 1 and 5. Nonetheless, PASC always performs $check(v)$ comparisons in software. Having duplicated state variables for comparison doubles the memory footprint of the process state.

SEI leverages the presence of ECC to minimize memory overhead, but it requires a different algorithm to deal with the fact that a variable and its replica are not stored separately (see Figure 1d). During the first execution, SEI takes a snapshot O of the original value of X. The new reset phase restores the snapshot and stores the state updates in N. The second execution is then executed again on the original state. This snapshot-and-reset strategy introduces additional hardening-specific data structures, which could be corrupted by ASC faults. We discuss next how to deal with these corruptions. Hardening of multithreaded applications, another major improvement of SEI over PASC, is also covered in Section 4.2.

4.2 SEI algorithm description

We now present the SEI algorithm and informally argue its correctness in the presence of some ASC faults. For a more detailed description and a complete correctness proof, we refer to our technical report [11].

Overview. SEI modifies all the three phases executed by event-driven threads (see Section 3). Hardening the dispatching and output phases consists of attaching a message replica c to every message m, *e.g.*, in the form of a CRC. The core challenge is hardening the event handling phase. SEI replaces the original event handler with a *hardened event handler* consisting of five phases: an initialization phase (I), a first-execution phase (E1), a reset phase (R), a second-execution phase (E2), and a final validation phase (V) (see Figure 2).

Figure 3 describes these phases in more detail. The operations in Phases I, R, and V are under the control of SEI and independent of the event handler code. Phases E1 and E2 execute the original event handling code of the application; SEI only inserts additional checks and operations according to Rules R1-R9. Some rules are actions taken before or after statements of the original

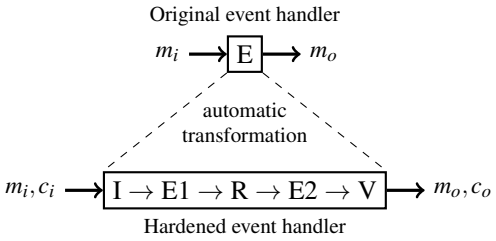

Figure 2: SEI transformation of an event handler E.

handler (denoted using *before/do* and *after/do* constructions), others are statements of the original handler that are replaced (denoted using *replace/with* constructions).

Phase I initializes SEI-internal data structures if the input message m_i matches its CRC c_i, otherwise it discards m_i. Phase E1 updates the state and generates one or more output messages m_o. Phase R restores the state prior to the event handler execution E1. Phase E2 updates the state again and generates the CRC c_o sent with each message m_o. Phase V compares the state updates of E1 and E2, aborting the process if they mismatch.

We start by describing SEI for single-threaded applications and then extend the description to include our multithreading support.

Internal data structures. During Phase E1, SEI takes a snapshot of the current value of variables before they are updated for the first time and stores them in a *snapshot buffer O* (Rule R2). Before Phase E2, it runs a reset phase (Phase R), which stores the current state of the variables updated by Phase E1 in a *new-value buffer N*, and restores the original values in *O*. The new value buffer is compared during Phase V with the *update buffer U*, which is created during Phase E2 (Rule R6), to make sure that the same set of variables is modified.

ASC faults during event handling. SEI executes event handlers twice to detect computation errors. To give some intuition on how SEI detects faults, say a process executes all phases in order I → E1 → R → E2 → V, and the internal data structures used by SEI for hardening are not corrupt. By the fault frequency property (Section 3), we have at most one fault during these steps, so either Phase E1 or E2 is fault-free. The integrity checks of Rules R1 and R5 guarantee that a fault-free execution of E1 or E2 does not read values corrupted by a fault.

Although SEI does not require ECC, an implementation of SEI can leverage ECC in the memory hierarchy because R1 (and its counterpart R5 in Phase E2) can be efficiently checked by the hardware (we perform the check when reading the variable). If the thread reaches Phase V and executes it correctly, then the latest state updates and output messages are correct. Note that even if a fault corrupts one of the two event handlers and skips

Phase I

```
1 if CRC(m_i) ≠ c_i then
2     discard m_i and return
3 initialize SEI variables
4 increment checking barrier
```

Phase R

```
1 store current values of updated variables in N
2 restore original values from O using write(v, val)
```

Phase E1 (generate output m_o)

```
R1: before read(v) do
        if ¬check(v) then Abort
R2: replace write(v, val) with
        if first write to v then
            add old value of v to O
        write(v, val)
R3: before acquire lock L do
        add L to Q
R4: replace release lock L with
        if not holding L then Abort
```

Phase E2 (generate CRC c_o for m_o)

```
R5: before read(v) do
        if ¬check(v) then Abort
R6: replace write(v, val) with
        add v to U
        write(v, val)
R7: after write(v, val) with v ∈ m_o do
        if last write to v then
            append val to CRC c_o
R8: before acquire lock L do
        add L to Q
R9: replace release lock L with
        if not holding L then Abort
```

Phase V

```
1 if ¬check(U) or ¬check(N) or ¬check(v) for each v in N then Abort
2 if CRC(m_i) ≠ c_i then Abort
3 if CRC(m_o) ≠ c_o for each output message m_o then Abort
4 if current state inconsistent with the updates in N then Abort
5 if N and U do not contain the same set of variables then Abort
6 release all locks in Q
7 wait at checking barrier
```

Figure 3: Rules and actions for each of the phases of a hardened handler. The pairs *before/do*, *after/do*, and *replace/with* indicate operations of the original event handler that are intercepted. The Rules R2 and R6 are described in more detail in Figure 4.

Phase V, the recipient of the message can still use m_o and c_o to verify that the outputs of $E1$ and $E2$ match.

Control-flow gates. SEI can handle much more complex fault scenarios. Due to control-flow faults, a sequence of instructions may be executed multiple times, in full or in part, or may be skipped altogether. We use *control-flow gates*, similar to PASC, to detect incorrect control-flow jumps from one phase of the hardened event handler (Figure 2) into another. We show a simplified example in Figure 4a, in which we use a control-flow variable *cf* (initially set to FALSE) to detect control-flow faults jumping from some phase P_1 to its subsequent phase P_2 and the other way around. Both phases, and thus Lines 1 and 5, represent multiple instructions. If a fault jumps from some instruction in P_1 to some instruction in P_2, then *cf* is not TRUE at Line 6, causing the process to abort. Likewise, if a fault jumps from some instruction in P_2 to some instruction in P_1, then *cf* is already TRUE at Line 2, causing the process to abort. Our technical report contains a more detailed description of gates and covers many more control flow scenarios, including cases where faults corrupt the control-flow variables.

Corruption of SEI-internal data structures. SEI also tolerates faults corrupting SEI-internal data structures. We now discuss two example scenarios. First, consider

a) Control-flow gate example

```
   // one phase
1  some phase P1
2  if read(cf) = TRUE then
3  |  Abort
4  write(cf, TRUE)
   // another phase
5  subsequent phase P2
6  if read(cf) = FALSE then
7  |  Abort
```

b) Detailed hardening Rules R2 and R6

```
R2: replace write(v, val) with
1   if read(O_c(v)) = FALSE then
2   |  write(O_v(v), read(v))
3   |  if read(O_c(v)) = TRUE then Abort
4   |  write(O_c(v), TRUE)
5   |  if read(v) ≠ read(O_v(v)) then Abort
6   write(v, val)
7   if read(O_c(v)) = FALSE then Abort

R6: replace write(v, val) with
1   write(U(v), TRUE)
2   write(v, val)
3   if read(U(v)) = FALSE then Abort
```

Figure 4: Control-flow gates and details of SEI-internals.

the example of a *cross-execution propagation*: if a fault occurs during the first execution (*i.e.*, Phase E1) and a variable update is not recorded in O, then the update will not be reset. As a consequence, the second execution (*i.e.*, Phase E2) could run incorrectly. Figure 4b shows the detailed pseudocode of Rule R2. We split the data structure O in two maps O_v and O_c. For each variable v updated during Phase E1, a variable $O_v(v)$ keeps the old value of v while a variable $O_c(v)$ determines (with a boolean) whether the old value of v is already contained in $O_v(v)$ or not. Technically, $O_v(v)$ and $O_c(v)$ are also variables in the process state. Before updating a variable v, the process stores the current value of v into $O_v(v)$ if O_c does not *contain* v yet, *i.e.*, $read(O_c(v)) = \text{FALSE}$. After updating v, the process checks that now O_c contains v in Line 7. This guarantees that if v is updated in Line 6, then by fault frequency either (a) no fault occurs before the update in the current (hardened) event handler execution, so the original value of v is stored into $O_v(v)$, or (b) no fault occurs after the update in the current (hardened) event handler execution, so the process crashes if it detects the absence of an entry for v in O_c. A similar pattern is used to update variable U (see Rule R6 in Figure 4b) and during Phase R to update the map N (see technical report for details).

As another example, consider a *snapshot buffer corruption*: if a variable v is updated multiple times during Phase E1 and a fault occurs, then a newer value of v could be written into $O_v(v)$ instead of the original value. To deal with this problem, SEI first writes the value of v into $O_v(v)$ in Rule R2 and later marks v as contained (Lines 2 and 4 of Figure 4b). Between these two operations, SEI checks that v is not contained in O_c yet, and aborts the process otherwise. Say v is updated a second time. After the first update of v, $read(O_c(v)) = \text{TRUE}$, otherwise the process would have aborted in Line 7; hence, the condition of Line 1 in the second update of v is false. If $O_v(v)$ is anyway overwritten in Line 2 during the second update, a fault must have changed the control flow to skip Line 1; by the fault frequency assumption, Line 3 executes correctly and aborts the process.

Computation scalability. Threads in a multithreaded application share the memory space of the process, but they can also have a set of private variables (stack and thread-local variables). Concurrently executing threads and sharing variables make single-threaded hardening techniques, like the one of Correia *et al.* [23], unsuitable for multithreaded applications. Consider two threads t_1 and t_2 that access the same set of shared variables. If a fault occurs, thread t_1 could write an incorrect value in a shared variable v. Thread t_2 could then read from v in the first execution of the event handler code without being able to detect the error through integrity check. Thread t_1 could then write an incorrect value into v again just before t_2 reads v in its second execution of the event handler code. The main consequence is that t_2 may experience a situation that, in a single-threaded setting, is equivalent to multiple state corruptions during the same (hardened) event handler execution.

The basic requirement of SEI to prevent this type of situations is that the threads of the original application access shared state only within critical sections protected by locks, as discussed in Section 3. Furthermore, in the presence of multiple locks, threads avoid deadlocks by acquiring and releasing the locks they need according to a predefined total order (or hierarchy).

SEI prevents error propagation across threads through shared state using three techniques: *deferred lock releasing*, *validated locking*, and *checking barrier*. Deferred lock releasing prevents error propagation among threads *as long as no two threads enter the same critical section*. A thread hardened with SEI postpones lock release operations during Phases E1 and E2 (Rules R4 and R9). The thread releases its locks only after Phase V, which guarantees that a thread only reads validated state updates from another thread. Deferred lock releasing results in longer critical sections, but this is not a problem for liveness since we target applications using lock hierarchies to avoid deadlocks.

Validated locking addresses situations when a control-flow error causes two threads to enter the same critical section. SEI ensures that the process crashes before any message is sent in such cases. During Phase E1, SEI records the locks a thread acquires (Rules R3 and R8). When SEI intercepts release operations in Phase E1 or E2, it verifies that the thread actually holds the lock (Rules R4 and R9). If the verification fails, then the process crashes.

Consider threads t_1 and t_2 entering the same critical section C. Let S be the process state when t_2 executes the first operation of C. Since t_1 and t_2 are in the same critical section, there must have been a fault during the execution of the current event handler of t_1 or t_2 before S. By fault frequency, there is no fault after S in the current hardened event handler execution of t_1 or t_2. Given S and C, there

```
while(1) {
  ilen = recv_msg_and_crc(imsg, &crc);
  // hardened event handler
  if (__begin(imsg, ilen, crc)) {
    do_something_here(msg);
    omsg = create_a_message_here(&olen);
    __output_append(omsg, olen);
    __output_done(); // finalize CRC
    __end();
  } else continue; // discard invalid input
  send_msg_and_crc(omsg, olen, __crc_pop());
}
```

Figure 5: Example of event loop and hardened handler.

is a unique set of locks that will be eventually released after C. At the state S, at most one of the two threads holds all necessary locks in the set. While releasing the locks, either t_1 or t_2 detects that it does not hold the necessary locks and crash the process. This verification step is still not sufficient, however. Say t_1 does not hold the necessary locks but t_2 does. Even if t_1 crashes the process upon exiting C, t_2 might still have time to exit C correctly and send an incorrect message before the crash.

The checking barrier is designed to prevent this last problem. It guarantees that if two threads execute a critical section concurrently, like t_1 and t_2 in the previous example, they do not send out any message before both have exited their critical sections and checked their locks. Each thread is associated to a concurrency counter, initially zero. When starting the execution of an event handler, a thread increments its own counter (Phase I). After comparing all state modifications in Phase V, the thread increments its counter again. If the counter value of any thread is odd, then it indicates that the thread (*e.g.*, t_2 in the previous example) might hold locks that have not been yet released and verified. After incrementing its counter, a thread takes a snapshot of the current counters for all threads and waits until all threads have either an even counter (they have released and verified all their locks and are ready to complete the execution of the event handler) or a counter higher than the snapshot (they have already started the next event handler execution). The checking barrier's caveat is the additional assumption that a single fault cannot make a thread skip the barrier increment (Line 4 of Phase I), enter a critical section in Phase E1 or E2 without acquiring a lock, and write an incorrect value onto a variable, because this would create an undetectable error for other threads. This scenario did not arise in our fault injection experiments.

5 SEI-hardening implementation

We now present `libsei`,[1] a library designed to automatically harden crash-tolerant distributed systems. `libsei` does not require re-developing the system from scratch, enabling existing code to be hardened with minimal effort, as we discuss in this section and in Section 6.

Hardening code with `libsei`. Hardening an event handler using `libsei` only requires: (i) marking the beginning and the end of an event handler using the macro functions `__begin()` and `__end()`; (ii) calling `__output_append(var, var_len)` to indicate that a variable `var` is added to the current output messages; (iii) calling `__output_done()` to indicate that the output message is complete and its CRC can be finalized and added to the output buffer; (iv) appending CRCs to output messages after retrieving them by calling `__crc_pop()`; and finally (v) starting the compiler as described below. The developer must enclose all operations modifying the process state with `__begin()` and `__end()`. During run time, the event handler executes twice with mechanism similar to `setjmp/longjmp` [33] implemented in `libsei`. Dispatching and output phases are external to `libsei` and do not require interaction with the library. Note that `__output_append()` and `__output_done()` can be called multiple times to generate multiple output messages in one handler.

Figure 5 shows the pseudo-code of a typical event-based process. The functions provided by `libsei` are prefixed with "`__`"; all remaining code is part of the pre-existing code base that needs to be hardened. Apart from adding some annotations and adding CRCs to messages, which is good practice anyway, there is not much a developer needs to do for hardening.

When the hardened system runs with multiple threads, the function `__barrier()` returns false if the thread should wait for another thread to complete the execution of its handler. The developer is responsible for calling `__barrier()` and blocking the output while it returns false. In Section 6, we discuss how to mitigate the overhead of blocking on the checking barrier.

Development effort. Scaling to large or existing code bases requires minimizing the development effort of using `libsei`. A major challenge is storing snapshots and state updates transparently. Instead of letting the developer notify the hardening library about state accesses, `libsei` automatically intercepts memory operations using a compiler transformation available out-of-the-box. In particular, we use transactional memory (TM) support of GCC, which is available from version 4.7 [27]. The TM compiler option redirects all memory operations within `__begin()` and `__end()` markers to a standardized application binary interface (ABI) [34]. Note that `libsei` provides the ABI but does not implement or rely on a TM algorithm. `libsei` merely executes procedures that store snapshots, state updates and perform validation, as described in Section 4.

`libsei` allows the developer to choose what event

[1] `http://bitbucket.org/db7/libsei`

handlers are and to protect only the important variables. For example, if a piece of code only manipulates a performance statistics variable, the developer might decide to keep the code outside any event handler since the variable does not contain critical data for the application safety. Also, `libsei` supports local handlers, helping the developer to call event handlers without explicitly receiving a message by marking the event handler with `__begin_nm()`, taking no messages as argument.

`libsei` internals. `libsei` tracks lock acquisitions and memory management by wrapping the `pthread_mutex` interface, `malloc()` and `free()`. After calling these functions in Phase E1, `libsei` saves the arguments and return value of the calls in a queue. In Phase E2, after checking the arguments to be the same as in Phase E1, it returns the values in the queue to the caller. Deallocations, similarly to lock releases, are postponed to the end of Phase V. In general, any function performing an external action – *e.g.*, sending a message – called inside an event handler has to be wrapped since it will be executed twice; the compiler terminates with an error otherwise. Among others, `libsei` currently wraps `sendto()` and `sendmsg()`, postponing their calls until the end of the second execution. No wrapper is necessary for external actions performed outside the event handler.

By default, `libsei` relies on memory error detection codes to keep variable replicas and execute *check* operations. This allows us to nearly eliminate the CPU and memory overhead of these operations.

6 Hardening real-world code bases

We have hardened two applications implemented in C: `memcached` and Deadwood. `memcached` is a popular multithreaded in-memory key-value cache [24], highly optimized for performance, that exposes a `get/set` interface to remote clients. `memcached` is essentially a large hashtable with an LRU eviction logic with linked lists to evict items. Deadwood is the single-threaded recursive DNS resolver of MaraDNS [49]. We have used `memcached` 1.4.15 and Deadwood version 3.2.05.

There are three main steps to harden a code base.

Step 1: Event handlers annotation. The initial challenge is choosing the right code lines to introduce the event handler markers. A good understanding of the code base is necessary to determine what state is persistent across the processing of multiple requests. In `memcached`, we marked 8 event handlers and added 7 lines related to the CRC of messages. More than 120 functions were automatically instrumented. In Deadwood, we marked 2 handlers and added 8 lines of code. More than 170 functions were automatically instrumented.

Step 2: Code base adaptation. Instrumentation is particularly simple in distributed systems that are logically organized as a collection of event handlers. These are common and Deadwood is a good example; we had to adapt only 2 code lines of Deadwood, moving a buffer to the heap to enable the reset of updates. Standard distributed computing algorithms such as the ones for state machine replication are typically specified and implemented as event-based algorithms as well. In some distributed system implementation, however, identifying a clean event-based pattern may be more challenging. Hardening required modifying and adding about 60 code lines to `memcached` because it does not always follow the pattern "dispatching, handling, output". One example is when an event handler of a `get` request retrieves an item: after sending the content of the item back to the client in the output phase, `memcached` decrements the reference counter of the item, which, being part of the state, should also be modified in hardened handlers. For such cases we have used local event handlers (see `__begin_nm()` in Section 5).

Another issue is that SEI currently only supports lock-based synchronization (see Section 4). The slab allocator of `memcached`, for example, uses *ad hoc* synchronization, so we disabled it for the hardened version. We left it enabled for the original version, however.

Step 3: Performance tuning. In some cases, the TM compiler might "over-protect" the code from the SEI's point of view. In Deadwood, dozens of strings are allocated and freed in the scope of a single handler; although these strings are in the heap memory, they are local variables of the handler and do not have to be protected. The developer can inform `libsei` to ignore writes into a region of memory, *e.g.*, into a string, by calling `__ignore_addr(addr, size)`. Moreover, if a complete function only modifies local variables, the instrumentation of the function can be disabled by declaring it with the `SEI_LOCAL` attribute.

To mitigate the effect of the checking barrier on the system scalability, the developer can adapt the system to handle other requests while a thread is waiting for other threads to complete the execution of concurrent event handlers. In `memcached`, a thread always serves another connection if sending a message would block the thread on the socket. We consequently fake a "would-block" case when a thread has to wait for the barrier. The caveat of this solution is the further 40 lines of code added to `memcached`. Alternatively, one can disable the barrier altogether, allowing threads to complete the handler execution without waiting for other threads. This solution requires no additional code change, but assumes locks cannot be skipped by ASC faults. We have implemented and evaluated both approaches and report results next.

Group	Fault	Description
CF	CF	IP register changes (control-flow fault)
DF	WREG	register value changes after it is written
	WVAL	memory value changes after it is written
	WADDR	calculated address changes before write
	RADDR	calculated address changes before read
RD	RREG	register value changes before it is read
	RVAL	memory value changes before it is read

Table 1: Fault types supported by our tool.

Group	Variant	Undetected	Det/SEI	Det/other	Total
CF	mc	9.66%	-	90.34%	6690
	mc-sei	0.06%	14.70%	85.23%	6515
	mc-sei-dup	0.00%	9.87%	90.13%	6594
DF	mc	44.18%	-	55.82%	15180
	mc-sei	0.15%	57.55%	42.29%	20264
	mc-sei-dup	0.00%	45.81%	54.19%	15991
RD	mc	33.04%	-	66.95%	10614
	mc-sei	0.52%	46.78%	52.70%	11508
	mc-sei-dup	0.00%	49.13%	50.87%	11442

Table 2: Errors classified in undetected, SEI-detected, and detected with other mechanisms. Total errors out of 8,000 executions for each fault-variant combination.

7 Fault coverage evaluation

The evaluation of SEI comprises two parts: fault coverage and performance. In this section, we report our fault coverage results; the performance results appear in Section 8. In the interest of space, we focus in this paper on the `memcached` results and briefly mention Deadwood experiments only to reinforce these results. Our technical report [11] contains all results for Deadwood.

To assess fault coverage, we performed two groups of experiments. First, we perform an extensive software fault injection campaign. Our goal is to determine (1) whether SEI effectively guarantees error isolation; and (2) how memory and computation scalability affect fault coverage. The second part consists of hardware fault injection using the dynamic voltage scaling of a processor. Our goal here is to collect evidence that our approach can indeed detect and isolate real, physically induced faults.

7.1 Software fault injection

Setup and methodology. In our fault injection experiments, we follow the approach of Basile *et al.* [8] and Correia *et al.* [23] injecting single bit flips. We have implemented a tool with Intel's Pin dynamic binary instrumentation framework [41] to inject faults during runtime. Our tool can inject three groups of faults described in Table 1. A control-flow (CF) fault flips a bit of the instruction pointer. A fault in the data-flow (DF) group affects the computation: WREG and WVAL represent incorrectly computed values that are respectively written into a register or a memory location, *e.g.*, an addition that results in a wrong value and is stored in a register; WADDR and RADDR represent computational errors while calculating an indexed address for reading or writing from memory. Finally, a fault in the RD group directly corrupts a register (RREG) before being used or a memory location (RVAL) before being read.

Field studies show that most memory faults are detected by ECC [32, 50]. Injected RVAL faults, however, automatically overwrite both, the value and its ECC. Hence, RVAL faults represent worst-case scenarios in which the ECC memory is not able to detect data corruption as assumed by corruption coverage (see Section 3).

To speed up our experiments and make the results reproducible, we have modified `memcached` to read commands from an input-trace file and write responses into an output-trace file by wrapping functions reading from and writing to sockets. To compare the output trace, we first create a golden run output-trace file. We perform two sets of experiments. The first set studies the fault coverage of SEI and the effects of leveraging hardware error detection codes in the implementation. We run, with a single thread, the unhardened `memcached` (mc), the SEI-hardened variant (mc-sei) with hardware error detection codes, and a further SEI-hardened variant (mc-sei-dup) with duplicated state assuming no error detection codes in hardware. The second set of experiments investigates whether the computational scalability aspect of our implementation affects the fault coverage. In this set, we run mc-sei and mc-seil with 4 threads; mc-seil has the checking barrier disabled and assumes that locks are not skipped. We perform 8,000 executions for each fault type and each single-threaded variant, with a subtotal of 64,000 executions for the DF group, 24,000 for the RD group, and a total of 168,000 executions (see Table 2). For the multithreaded experiments, we perform a total of 80,000 executions (see Table 3).

In each run, one fault is injected at a randomly selected instruction inside or outside the event handler including shared libraries; Pin cannot, however, instrument instructions inside syscalls. A fault that causes a trace deviation, *e.g.*, an unexpected message or a shorter trace, produces a *manifested error*. The errors we report are all manifested, consequently we refer to them as just errors henceforth.

Fault coverage and memory scalability. We initially experimented with a single thread to observe the effects of faults without the effects of concurrent access. Table 2 summarizes the results of our fault injection experiments with a single thread. The right-most column shows, for each fault-variant combination, the total number of errors out of the total of each group. Errors are detected or undetected, shown as percentage of the total number of errors. *Undetected* errors are *corrupt* output messages

that cannot be detected by the client. They correspond to error propagation scenarios where the error isolation property is violated. Detected errors are further divided into *det/SEI*, *i.e.*, errors detected and isolated by `libsei`, for example, crashes initiated by the library or invalid messages detectable at the client; and *det/other* errors, *i.e.*, errors detected or isolated by other mechanisms, for example, crashes due to segmentation fault or assertions, infinite loops, and also error messages or partial messages detectable at the client. Note that each error propagation percentage is relative to the total observations in each fault group. They do not express probabilities since the real frequency of CF, DF, and RD faults is unknown.

Hardening `memcached` drastically decreases the undetected errors. The native mc variant shows from 9% up to 44% undetected errors, depending on the fault group. In contrast, the mc-sei variant shows at most 0.15% undetected errors for DF faults and 0.52% for RD faults. The latter result indicates SEI is also resilient to fault scenarios where the ECC memory does not detect data corruption. Hardening Deadwood shows similar trends, reducing undetected errors, for example, from 32.38% down to at most 0.12% for the DF group.

Like PASC [23], mc-sei-dup uses software-duplicated state and detects all injected faults. As this work focuses on the use of hardware error detection, we now analyze how errors manifest specifically on the mc-sei variant.

Detected non-silent errors. Since hardening cannot guarantee fail-stop behavior, some errors are non-silent: clients perceive them as unexpected messages. A message is *invalid* if the message CRC does not match the message payload. From 0.7% up to 3.4% of the errors in mc-sei are invalid messages, representing the majority of non-silent errors. Some messages also arrive truncated at the client, *e.g.*, when `memcached` crashes before writing the complete message out. Interestingly, `memcached` itself produces error messages, for example, when a fault makes `memcached` think it is out of memory. Truncated and error messages constitute up to 1.2% of the errors. As shown in Table 2, undetected errors (*i.e.*, corrupt but valid messages) represent up to 0.52% of the errors and are the only cases that can violate error isolation. We now study these cases in detail.

Undetected errors analysis. Analyzing the log files of our experiments, we identified pointer corruption as the major source of undetected errors in mc-sei. Leveraging hardware error detection, as SEI does, has the side effect that variables and their replicas (the ECC data) are stored in the same memory location and accessed together by the processor. A fault corrupting a pointer to a variable in an undetectable manner causes both, the variable and its replica, to become corrupt, invalidating the fault diversity assumption. It is consequently a type of fault not covered by our fault model. The very low overhead of `libsei` and the results presented above (a drop of undetected errors from 44.18% to only 0.15%) are encouraging, however. Note also that using software replication overcomes this problem because we use two separate pointers for the value and its replica. Using software replication constitutes a trade-off between memory footprint and fault coverage (see mc-sei-dup in Table 2).

Group	Variant	Undetected	Det/SEI	Det/other	Total
CF	mc-sei	0.02%	13.78%	86.20%	6366
	mc-seil	0.05%	12.84%	87.11%	6330
DF	mc-sei	0.16%	58.58%	41.26%	19484
	mc-seil	0.28%	58.61%	41.11%	19088

Table 3: Errors for 4-threaded executions classified in undetected, detected with SEI, and detected with other mechanisms. Total errors out of 8,000 executions for each fault-variant combination.

To understand a typical scenario of error propagation, consider the following instructions, which are executed upon completion of sending a reply:

```
// item *it = *(c->icurr);
mov    (%rax),%rax
mov    %rax,-0xe0(%rbp)
```

After replying to a `get` request, `memcached` decrements the reference counter of the retrieved item. The object `c` is the connection, and `*(c->icurr)` is the address of the retrieved item, which is kept in the hashtable. The first instruction stores the address of the current item, `*(c->icurr)`, into register `rax`. The second instruction moves the address into the stack, *i.e.*, into the target address `-0xe0(%rbp)`. In our logs, a WADDR fault flipped the calculated address, making the `mov` operation write the pointer just after the stack. The execution proceeded to decrement the reference counter, which is executed in a hardened local event handler. The pointer used, however, was the wrong pointer because the address `-0xe0(%rbp)` still pointed to an old item in the hashtable. The old item had its reference counter decremented and was freed since its reference counter reached zero. The memory location was later reused for another entry of the hashtable, resulting in two item entries (keys) pointing to the same item object in memory incorrectly.

Computational scalability effects. Table 3 shows the results for our multithreaded experiments. The results indicate that (1) multithreaded executions do not present more undetected errors than single-threaded executions; and (2) although mc-seil assumes locks cannot be skipped, it does not show substantially more undetected errors than mc-sei. In particular, CF faults, which can potentially jump over locks, resulted in less than 0.1% of undetected errors in both variants.

Variant	Undetected	Det/SEI	Det/other	Total
mc-sei	0	11	457	468
mc	4	-	464	468

Table 4: Errors when undervolting CPU.

7.2 Hardware fault injection

Software fault injection can reproduce fault cases very precisely, making it easier to analyze and understand failures. However, there is the risk of introducing a bias, and consequently, we have also used hardware fault injection to reproduce realistic and unbiased failure scenarios.

We perform hardware fault injection by using the dynamic voltage and frequency scaling (DVFS) support of an AMD FX based multi-core CPU (Bulldozer). DVFS can be used to undervolt cores, reducing the voltage below the predefined value while keeping the frequency constant. The scenario of our experiments could be the effect of misconfiguration of power-saving options or of a power supply failure. Note that future microprocessors are also expected to run at lower voltage, thus increasing the likelihood of data corruption [15].

We experimented with variants mc and mc-sei running a single thread. After launching the application, we lowered the voltage between 100 and 150 mV of the nominal CPU voltage (1.225 V). Table 4 shows the outcome of 936 observations. The application often crashed in at most 40 seconds of execution. In addition to crashing, the machine froze very often, explaining the reduced number of experiments performed.

The vast majority of errors were crashes caused by segmentation faults, invalid instruction errors, and other errors detected by the operating system. In mc-sei, 2.35% of the errors (11 cases) were detected by `libsei`, and we observed no undetected errors. In mc, 0.85% of the errors (4 cases) were undetected. Although not conclusive, the experiment indicates that (1) undetected errors, *i.e.*, corrupt messages, can happen due to hardware faults; and (2) some of these faults manifest as ASC faults and are successfully detected by `libsei`.

8 Performance evaluation

Setup and methodology. We run the `memcached` with a hashtable of 2 GB on a 12-core 2.66 GHz Intel Xeon X5650 machine (Linux 3.8 kernel). We use 8 client machines with a similar configuration (8-core 2 GHz Xeon) connected via Gigabit Ethernet. Each client machine runs one instance of Facebook's `mcblaster` workload generator [38]. Each `mcblaster` instance measures averages of the throughput and response time for 60 s.

The workload can be configured with *value size* in bytes. One client machine with 64 connections is started

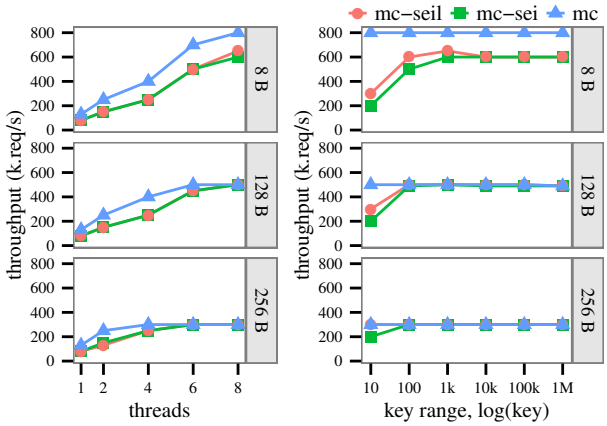

Figure 6: Throughput of get requests varying threads (with key range of 1000) and key range length (with 8 threads) for different value sizes (8, 128 and 256 bytes)

for each `memcached` *thread*. Clients randomly select (using uniform distribution) the next key to be issued from the integer set $\{1,\ldots,K\}$ where K is called *key range*. Finally, the *load* is the aggregated number of requests per second issued by all clients. Clients mainly issue `get` requests since they represent the vast majority of operations in typical workloads [5, 45, 52].

We consider the following `memcached` variants: mc, mc-sei and mc-seil. Stock `memcached` has an important bottleneck due to a global lock protecting the LRU eviction list, *i.e.*, `cache_lock`, which is known to affect scalability [5]. We have improved this bottleneck to increase scalability by having all our variants of `memcached` acquiring the `cache_lock` with `trylock()`, and only updating the list if there are no concurrent updates. Even with this bottleneck improvement, mc still does not scale above 8 threads, so we limit our experiments to up to 8 threads. Finally, to avoid modifying the workload generator, the hardened `memcached` variants compute 32-bit CRCs as prescribed by the algorithm, but do not send them along with messages. The expected performance impact of 4 bytes of CRC is negligible when added.

Deadwood is a single-threaded server. It follows a similar setup, but with up to 20 client machines running `nsping` to query the IP of 100 popular websites.

Computation and memory scalability. Figure 6 (left) represents the scalability limit for `memcached` when varying the number of threads from 1 to 8. The y-axis depicts the maximal throughput that can be achieved while keeping the average response time across all requests below 1 ms, a realistic response time target for `memcached`. We also vary the value size from extremely small messages (8 B) to medium messages (256 B). With larger value sizes, fewer threads are necessary to achieve the maximal throughput with any variant; all variants

achieve their maximal throughput with 8 threads. With value sizes larger or equal to 128 B and 8 threads, mc-sei and mc-seil show negligible throughput overhead. With a value size of 8 B and 8 threads, the overhead is 25% for mc-sei and 20% for mc-seil. mc-seil shows a lower overhead than mc-sei due to the disabled SEI's barrier.

Figure 6 (right) depicts the maximal throughput achieved when varying the key range and the value size with 8 threads. Few keys introduce contention between the threads, since they access the same buckets, acquiring at least 2 locks per request. Critical sections become longer due to hardening. Consequently, the scalability with queries spanning very few keys, *e.g.*, 10 keys, is limited. Such scenario could also represent a workload with a few hot keys. We expect, however, a `memcached` instance to host and serve many thousands of different keys. As we distribute the workload across more keys, there is less contention and consequently more opportunity for concurrent execution. The overhead with 1 M keys and 8 B value sizes, for example, is about 25% for both mc-sei and mc-seil. The overhead becomes negligible with larger value sizes and more than 100 keys.

Regarding memory overhead, each thread requires about 30 KiB for hardening-related data structures.

Single-thread scenarios. `libsei` is designed to amortize its overhead with the number of threads. Multi-threading can release pressure on the CPU, making it more likely for the system to become network bound. We now consider single-threaded scenarios with Deadwood and `memcached` running with one thread – see [11] for details on these experiments.

When the system is not overloaded, the response time overhead of mc-sei is small. For example, at a load of 10 k.req/s, the difference of response time between mc and mc-sei varies from 2% to 7% depending on the size of the messages – we experimented with values of 8 B up to 8 KiB. In Deadwood, depending on the response size, hardening incurs an overhead from 13% up to 21% in the response time for loads of 1 k.req/s.

Deadwood becomes CPU bound very quickly. As a result, the throughput overhead under maximal request load reaches 40% to 50%. Using the timestamp counter of the processor, we measured the average number of cycles Deadwood consumes for a single request in the dispatch, handling, and output phases (averaged over 10,000 requests). Table 5 shows the percentage of CPU cycles spent in each phase relative to the native variant. The dispatch and output phases do not increase significantly with hardening. The hardened handling phase takes, however, 2.4 times the number of the cycles of the native counterpart. This overhead is caused by the double execution of the event handler, by the code instrumentation, and by the checks in `libsei`. Since the duplicated part constitutes only 27% of the used cycles, the cycle-overhead of processing a single message by hardened Deadwood is only 41%.

Variant	Dispatch	Handling	Output	Total	Cycles
Native	41.75%	27.51%	30.74%	100%	83 k
SEI-hardened	42.71%	66.81%	31.47%	141%	117 k
Overhead	+0.94%	+39.30%	+0.73%	+41%	

Table 5: Average CPU cycles consumed for a single request relative to the native Deadwood variant.

In contrast, mc-sei is not likely to saturate even with a single thread. For 1 KiB large values, mc-sei has an overhead of 20%. For 4 KiB or larger values, both mc-sei and mc are network bound and show no significant difference in the throughput.

Overall, even in single-threaded scenarios we observed no more than 50% overhead. The low overhead is due to the hardening of application event handlers, but not the underlying software components, such as the operating system. SEI expects faults in these components to manifest as ASC faults, corrupting the application state or its messages. According to our fault injection experiments, SEI is sufficient and a "duplicate everything" strategy is not strictly necessary. We expect long event-handling phases, however, to induce higher overheads.

9 Conclusion

We have proposed a novel algorithm for ASC hardening, SEI, that can leverage mechanisms provided in hardware, such as error correction codes in memory modules, to minimize overhead. The exercise of hardening an existing system like a DNS server and `memcached` exposed a number of challenges, mostly related to deviations to the structure our algorithm expects. Yet, we were able to harden it with some reasonable amount of effort. SEI introduces a negligible overhead in applications that are not CPU-bound and is effective in avoiding error propagation. The residual error propagation observed in our fault injection results is due to pointer corruption, which SEI is vulnerable to when using hardware ECC. It is subject of future work to design new techniques or extensions that are able to overcome this limitation while using ECC memory for systems that are potentially more susceptible to pointer corruption.

Acknowledgments

This work was partially supported by the state of Saxony under grant of ESF 100111037 and by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden". We would like to thank our reviewers and our shepherd Richard Mortier for the useful feedback.

References

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

[2] AMAZON. New defective S3 load balancer corrupts relayed messages. https://forums.aws.amazon.com/thread.jspa?threadID=22709, June 2008.

[3] AMAZON. Odd data corruption during download. https://forums.aws.amazon.com/thread.jspa?messageID=86214, Apr. 2008.

[4] AMAZON. Single-bit corruption of a small percentage of S3 data. https://forums.aws.amazon.com/thread.jspa?messageID=262676, July 2011.

[5] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 53–64.

[6] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J. J., KHORLIN, A., LARSON, J., JEAN, LÉON, M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *5th Biennial Conference on Innovative Data Systems Research (CIDR)* (2011).

[7] BARTLETT, W., AND SPAINHOWER, L. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing 1*, 1 (2004), 87–96.

[8] BASILE, C., LONG, W., KALBARCZYK, Z., AND IYER, R. Group communication protocols under errors. In *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems* (2003), pp. 35–44.

[9] BEHRENS, D., FETZER, C., JUNQUEIRA, F. P., AND SERAFINI, M. Towards transparent hardening of distributed systems. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems* (2013), HotDep'13.

[10] BEHRENS, D., KUVAISKII, D., AND FETZER, C. HardPaxos: Replication Hardened against Hardware Errors. In *IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS), 2014* (Oct. 2014), pp. 232–241.

[11] BEHRENS, D., SERAFINI, M., ARNAUTOV, S., JUNQUEIRA, F., AND FETZER, C. Scalable error isolation. Tech. Rep. TUD-FI15-01-Februar 2015, ISSN 1430-211X, Technische Universität Dresden, Fakultät Informatik, Feb. 2015. http://bitbucket.org/db7/libsei.

[12] BEHRENS, D., WEIGERT, S., AND FETZER, C. Automatically tolerating arbitrary faults in non-malicious settings. In *Proceedings of the Sixth Latin-American Symposium on Dependable Computing (LADC)* (April 2013), pp. 114–123.

[13] BHATOTIA, P., WIEDER, A., RODRIGUES, R., JUNQUEIRA, F., AND REED, B. Reliable data-center scale computations. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware* (New York, NY, USA, 2010), LADIS '10, ACM, pp. 1–6.

[14] BORKAR, S. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro 25*, 6 (2005), 10–16.

[15] BORKAR, S., AND CHIEN, A. A. The future of microprocessors. *Communications of the ACM 54*, 5 (2011), 67–77.

[16] BORKAR, S., ET AL. Microarchitecture and design challenges for gigascale integration. In *MICRO* (2004), vol. 37, pp. 3–3.

[17] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 335–350.

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

[19] CASTRO, M., RODRIGUES, R., AND LISKOV, B. BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems 21*, 3 (Aug. 2003), 236–269.

[20] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 398–407.

[21] CONSTANTINESCU, C. Trends and challenges in vlsi circuit reliability. *IEEE Micro 23*, 4 (2003), 14–19.

[22] CORBETT *et al.*. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[23] CORREIA, M., FERRO, D. G., JUNQUEIRA, F., AND SERAFINI, M. Practical hardening of crash-tolerant systems. In *2012 USENIX Annual Technical Conference* (2012).

[24] DANGA INTERACTIVE, INC. `memcached` – a distributed memory object caching system. `http://memcached.org`.

[25] DINABURG, A. Bitsquatting: DNS hijacking without exploitation. In *Defcon 19* (2011).

[26] FENG, S., GUPTA, S., ANSARI, A., AND MAHLKE, S. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2010), ASPLOS XV, ACM, pp. 385–396.

[27] Transactional Memory in GCC. `http://gcc.gnu.org/wiki/TransactionalMemory`.

[28] GUPTA, A., YANG, F., GOVIG, J., KIRSCH, A., CHAN, K., LAI, K., WU, S., DHOOT, S., KUMAR, A., AGIWAL, A., BHANSALI, S., HONG, M., CAMERON, J., SIDDIQI, M., JONES, D., SHUTE, J., GUBAREV, A., VENKATARAMAN, S., AND AGRAWAL, D. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *VLDB* (2014).

[29] HAMILTON, M. *Software development: building reliable systems*. Prentice Hall Professional, 1999.

[30] HO, C., VAN RENESSE, R., BICKFORD, M., AND DOLEV, D. Nysiad: Practical protocol transformation to tolerate byzantine failures. In *NSDI'07: Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation* (2007).

[31] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of USENIX Annual Technical Conference* (2010).

[32] HWANG, A. A., STEFANOVICI, I., AND SCHROEDER, B. Cosmic rays don't strike twice: Understanding the nature of DRAM errors and the implications for system design. In *ASPLOS* (2012).

[33] IEEE Std 1003.1 and The Open Group Base Specifications, Issue 7. `http://pubs.opengroup.org/onlinepubs/9699919799/functions/setjmp.html`, 2013.

[34] INTEL. Intel Transactional Memory Compiler and Runtime Application Binary Interface, 2009.

[35] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., DAHLIN, M., ET AL. All about eve: Execute-verify replication for multi-core servers. In *OSDI* (2012), vol. 12, pp. 237–250.

[36] KERNEL BUG TRACKER. Data corruption with Opteron CPUs and Nvidia chipsets. `https://bugzilla.kernel.org/show_bug.cgi?id=7768`, Jan. 2007.

[37] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of ACM SOSP* (2007), pp. 45–58.

[38] KWIATKOWSKI, M. `mcblaster` - load generator for `memcached`. `http://github.com/fbmarc`.

[39] LAMPORT, L. The part-time parliament. *ACM Transactions on Computing Systems (TOCS) 16*, 2 (1998), 133–169.

[40] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4*, 3 (1982).

[41] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[42] MERIDETH, M. G., IYENGAR, A., MIKALSEN, T., TAI, S., ROUVELLOU, I., AND NARASIMHAN, P. Thema: Byzantine-fault-tolerant middleware for Web-service applications. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2005), SRDS '05, IEEE Computer Society, pp. 131–142.

[43] NIGHTINGALE, E. B., DOUCEUR, J. R., AND ORGOVAN, V. Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. In *Proc. of Eurosys* (2011), pp. 343–356.

[44] NIKIFORAKIS, N., VAN ACKER, S., MEERT, W., DESMET, L., PIESSENS, F., AND JOOSEN, W. Bitsquatting: exploiting bit-flips for fun, or profit? In *Proceedings of the 22nd international conference on World Wide Web* (2013), International World Wide Web Conferences Steering Committee, pp. 989–998.

[45] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 385–398.

[46] OH, N., SHIRVANI, P., AND MCCLUSKEY, E. Error detection by duplicated instructions in superscalar processors. *Reliability, IEEE Transactions on 51*, 1 (Mar. 2002), 63–75.

[47] PERRY, F., MACKEY, L., REIS, G. A., LIGATTI, J., AUGUST, D. I., AND WALKER, D. Fault-tolerant typed assembly language. In *ACM SIGPLAN Notices* (2007), vol. 42, ACM, pp. 42–53.

[48] REIS, G., CHANG, J., VACHHARAJANI, N., RANGAN, R., AND AUGUST, D. SWIFT: software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization* (Mar. 2005), pp. 243–254.

[49] SAM TRENHOLME AND OTHERS. Deadwood recursive DNS resolver. `http://maradns.samiam.org/deadwood`.

[50] SCHROEDER, B., PINHEIRO, E., AND WEBER, W.-D. DRAM errors in the wild: a large-scale field study. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems* (New York, NY, USA, 2009), SIGMETRICS '09, ACM, pp. 193–204.

[51] SHIVAKUMAR, P., KISTLER, M., KECKLER, S. W., BURGER, D., AND ALVISI, L. Modeling the effect of technology trends on the soft error rate of combinational logic. *International Conference on Dependable Systems and Networks* (2002), 389.

[52] VENKATARAMANI, V., AMSDEN, Z., BRONSON, N., CABRERA III, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., HOON, J., KULKARNI, S., LAWRENCE, N., MARCHUKOV, M., PETROV, D., AND PUZAR, L. TAO: How facebook serves the social graph. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 791–792.

[53] WOOD, A., JARDINE, R., AND BARTLETT, W. Data integrity in HP NonStop servers. In *Workshop on SELSE* (2006).