IEICE TRANS. INF. & SYST., VOL.E107–D, NO.11 NOVEMBER 2024

PAPER

Runtime Tests for Memory Error Handlers of In-Memory Key Value Stores Using MemFI

Naoya NEZU[†], *Nonmember and* **Hiroshi YAMADA**[†a)], *Member*

SUMMARY Modern memory devices such as DRAM are prone to errors that occur because of unintended bit flips during their operation. Since memory errors severely impact in-memory key-value stores (KVSes), software mechanisms for hardening them against memory errors are being explored. However, it is hard to efficiently test the memory error handling code due to its characteristics: the code is event-driven, the handlers depend on the memory object, and in-memory KVSes manage various objects in huge memory space. This paper presents *MemFI* that supports runtime tests for the memory error handlers of in-memory KVSes. Our approach performs the software fault injection of memory errors at the memory object level to trigger the target handler while smoothly carrying out tests on the same running state. To show the effectiveness of MemFI, we integrate error handling mechanisms into a real-world in-memory KVS, memcached 1.6.9 and Redis 6.2.7, and check their behavior using the MemFI prototypes. The results show that the MemFI-based runtime test allows us to check the behavior of the error handling mechanisms. We also show its efficiency by comparing it to other fault injection approaches based on a trial model.

key words: *memory errors, ECC-uncorrectable memory errors, fault injection*

1. Introduction

A memory error is an event in which the logical state of one or multiple bits is read differently from how it was last written. Memory errors may be caused by electrical or magnetic interference (e.g., due to cosmic rays) or by hardware problems (e.g., a bit being permanently damaged). A study has revealed that memory errors affect 32% of datacenter servers in a year and that 1.3% of such errors are not correctable [1]. Another study at Facebook showed that the memory error rate has increased as DRAM technology scales to smaller feature sizes [2], [3]. Moreover, emerging memory technologies such as 3D XPoint are likely to have a higher incidence of memory cell failures because their endurance is lower than that of DRAM [4], [5]. Although an error correction code (ECC) hardware module is the standard means of detecting and correcting such memory errors, multi-bit memory errors in the same word, which are rare but critical to computer systems, cannot be fixed by the widely used ECC memory based on single error correction-double error detection (SEC-DED) [6], [7]. Recent field studies on DRAM and SSDs [1], [8]–[10] have shown that detectable but uncorrectable media errors by ECC hardware frequently occur.

Since memory errors have a severe impact on in-memory key-value stores (KVSes), which manage data items in an associative manner and are typically used for data caching in modern web services [11]–[14], software mechanisms for hardening them against the memory errors are being explored [15]–[18]. In-memory KVSes such as memcached [19] and Redis [20] manage a tremendous number of items in memory, sometimes hundreds of gigabytes of memory in real-world platforms [21], [22], to achieve high throughput and low latency. Such a large memory footprint causes in-memory KVSes to face relatively more memory errors compared with other applications. In-memory KVSes affected by memory errors return wrong values or conduct their reboots for recovery, which are time-consuming due to hundreds of gigabytes of memory restoration [23]. To protect the in-memory KVSes against memory errors, software-based ECC [17] uses three different ECC codes to perform memory error detection and correction for metadata, keys, and values. The partial-surgery [15] prunes damaged memory objects and reconstructs the internal structures by using only the undamaged ones.

Testing of memory error handlers is mandatory. Since the memory error handlers update the internals of the running in-memory KVSes, the handlers' bugs cause the misbehavior and sometimes crashes of the KVSes. For example, the software-based ECC fixes the contents of the damaged KVs, and the partial-surgery prunes the damaged KVs and rebuilds other objects such as hash tables and free lists. Since bugs in the handlers can set incorrect KVs and break metadata structures, the KVS repaired by a buggy handler cannot run properly.

However, software mechanisms for memory errors pose a challenge in testing their memory error handling code. Due to its unique characteristics, it is hard to test the behavior of memory error handlers efficiently. First, the handling code is executed only when the target software faces memory errors in its address space. Such an event-driven code is hard to walk even with modern fuzzing testing [24]–[27] since the different types of KV commands, such as get and set, generated as inputs to in-memory KVSes do not lead to memory error handler execution. Second, the handlers depend on the memory objects. Although the software fault injection tool [28]–[32] is sometimes the only way to trigger the handling code execution by rewriting memory contents as memory errors, it takes quite a long time to check the target handler; typical fault injectors randomly choose mem-

Manuscript received January 25, 2024.
Manuscript revised May 28, 2024.
Manuscript publicized July 11, 2024.

[†]Tokyo University of Agriculture and Technology (TUAT), Koganei-shi, 184–8588 Japan.
a) E-mail: hiroshiy@cc.tuat.ac.jp

DOI: 10.1587/transinf.2024EDP7019

ory addresses and instructions for error injection. To test the target handler, its developers have to repeatedly perform the fault injection until the injector inserts a memory error to the corresponding memory object among various objects in the in-memory KVS. Lastly, restoring an in-memory KVS running state is so time-consuming that tests on the same state can be a non-trivial task. For example, in checking each behavior of memory error handlers for metadata, key, and value on the in-memory KVS with full items, the developers feed tens to hundreds of gigabytes of KVs into it every handler test.

This paper presents *MemFI* that supports runtime tests for the memory error handling code of in-memory KVSes. The key idea behind MemFI is to perform the software fault injection of memory errors at the memory object level and check the target handlers in a phase-based manner. Unlike conventional fault injectors, MemFI identifies memory objects in the address space and injects bit flips to the memory addresses within the target object, such as metadata and keys. The MemFI-based runtime test consists of three phases; *warm-up*, *injection*, and *check phases*. In the warm-up phase, a tester stores KVs to the in-memory KVS until its internals becomes one on which he or she wants to test, while specifying the memory addresses to inject memory errors in the injection phase. The check phase executes the fault-injected in-memory KVS and quickly reverts its state just after the warm-up phase. As the case studies, we integrate two memory error handling mechanisms [15], [17] into real-world in-memory KVSes, memcached [19] and Redis [20], and check their behavior with the MemFI prototypes. Our experimental results show that the MemFI-based runtime tests allow us to check the runtime behavior of the error handling mechanisms. To compare the MemFI-based test to other fault injector-based ones, we build a trial model that estimates the trial numbers of fault injections required for checking the behavior of the target handler.

This paper is substantially extends our previous work [33]. The main differences are that we (1) prototyped MemFI on Redis and performed MemFI-based runtime tests for partial-surgery and checksum-based logging for it, (2) compared the prototype to typical random fault injectors based on the trial model, and (3) discussed the motivation of runtime tests for memory error handlers and work related to our in more detail.

In summary, this paper makes the following contributions:

- We present MemFI that supports runtime tests for memory error handling code of in-memory KVSes. Compared with the existing software testing schemes, the approach is characterized as follows. MemFI allows us to test memory error handling code. MemFI triggers different memory error handlers. It is also a pure software approach that does not require any modification of the underlying hardware and in-memory KVS source code (Sect. 3).
- To test memory error handlers, MemFI offers three techniques; *memory object-level fault injection*, *error-firing control*, and *phase-based testing*. The memory object-level fault injection extracts the virtual addresses of objects and injects errors to the specified addresses. The error-firing control fires memory errors at various instructions accessing the specified addresses. The phase-based testing clearly decouples a phase for warming up the target in-memory KVS from software testing. In the MemFI-based runtime test, we perform various test trials, each of which consists of specifying the memory address to inject an error and accessing the error-injected address on the warm-up in-memory KVS (Sect. 4 and 5).
- We apply MemFI into two real-world in-memory KVSes on Linux 5.10.0; memcached 1.6.9 and Redis 6.2.7. We implement the partial-surgery [15] and software-ECC [17] on memcached, and check their behavior with the memcached-integrated MemFI. We also implement the partial-surgery and extend the KV saving feature to be memory error-aware on Redis, and test their code with MemFI (Sect. 6 and 7).
- To show the efficiency of MemFI-based runtime tests, we build a trial model that estimates how many fault injection trials are required to comprehensively check the target error handlers. Using it, we compare our prototypes to other fault injectors (Sect. 8).

2. Motivation

2.1 In-Memory KVSes under Memory Errors

Modern memory devices such as DRAM are prone to errors that occur because of unintended bit flips during their operation [1], [3], [34]. There are two types of memory errors: hard (or recurring) and soft (or transient). Hard errors are caused by physical device defects and environmental factors such as humidity, temperature, and utilization. Typically, hard errors affect multiple bits. On the other hand, soft errors transiently occur as a result of charged particle emissions from chip packages or the atmosphere. In particular, some researchers and practitioners have performed field studies on the characteristics of memory errors occurred in real-world data centers [1], [34]–[36], while others have investigated the behavior of software and hardware when memory errors occur [37]–[40].

While ECC hardware modules that detect and correct memory errors are widely employed to mitigate such errors, ECC-uncorrectable memory errors that can be detected but cannot be fixed by ECC modules affect the running applications. The current ECC modules cannot always fix memory errors. SEC-DED and DEC-TED correct only single and double-bit errors, thus failing to fix three or more bit errors. Chipkill [41] can correct memory errors in 1/8 chip and detect the errors in 2/8 chips but cannot handle memory errors that damage 3/8 chips. Some researchers have studied such multi-bit memory errors [37], [42].

The standard way to handle memory errors is to restart the affected applications. If an administrator detects a memory error in the address space of the running process, he or she relaunches the process if necessary. This reboot-based recovery implicitly assumes that target applications are stateless so that their memory objects can be quickly restored after the reboot. Since in-memory KVSes manage many memory objects, such as KVs and the running states on their memory space, rebooting them loses all memory objects. Restoring the memory objects is costly and leads to significant performance degradation. The restarted in-memory KVSes perform at a level much lower than their peak performance until the memory objects are restored. The common way to restore memory objects is to access a backing store like local/shared storage or replica just after the in-memory KVS fails. The restarted in-memory KVS fetches the KVs from the sources to store them in its in-memory cache. This is widely known as a time-consuming task [23].

2.2 Testing Handlers for Memory Errors

In this paper, we try to answer the following question: *How can we test the memory error handlers of in-memory KVSes efficiently?* Expensive hardware modules for memory errors are not preferable and sometimes even unacceptable because of their high power consumption and design complexity [43], [44]. The researchers have studied software-based approaches to enforcing in-memory KVSes to survive memory errors [15]–[18]. The partial-surgery [15] allows in-memory KVSes to prune damaged memory objects and reconstructs their internal structures by using only the undamaged ones. Software-based ECC [17] uses three different ECC codes that provide more powerful memory protection than regular hardware-based ECC does for metadata, keys, and values, while Ring [18] leverages r-way replication and Reed-Solomon coding for the integrity of the target KVs. NEMESIS [45] and FlipBack [46] are compiler-based. NEMESIS runs three versions of computations and detects soft errors by checking the results of all memory write and branch operations, and FlipBack leverages compile-time analysis and program markup to identify data critical for control flow and enables selective replication of computations by the runtime system. These motivate us to explore a way to support the runtime tests of the error handlers.

Testing the mechanisms for memory errors is non-trivial due to the following characteristics of their code and in-memory KVSes. These motivate us to explore a way to support the runtime tests of the error handlers.

The memory error handlers are event-driven: Their code is executed only when the target software accesses memory objects damaged by memory errors, and such code is hard to execute even with fuzzing testing that forces the target software to walk its as many code paths as possible by feeding different types of KV commands into it. For example, the recovery logics of the partial-surgery and the compiler-based approaches never run until the object damaged by a memory error is accessed.

The memory error handlers depend on the memory objects: The handlers perform recovery specific to each type of memory objects. When memory errors damage a KV, the partial-surgery-aware Redis removes only the KV from its chain hash. On the other hand, when memory errors affect a KV metadata named dictEntry that has hash chain pointers, it reconstructs its hash chain since the successors in the damaged metadata cannot be accessed. The software fault injection is an effective and sometimes only way to trigger the handling code by rewriting the memory contents as memory errors. However, almost all fault injectors randomly choose memory addresses for error injection. Hence, to test an error handler, the developer must repeatedly perform fault injections until the tool inserts a memory error to the address in the corresponding object.

Restoring the running states of in-memory KVSes is time-consuming: Modern in-memory KVSes typically utilize tens to hundreds of gigabytes of memory and consist of various objects. For example, memcached's internals consists of size-based slabs, its metadata named slabclass_t, LRU- and free-lists in each slabclass, a chain-based hash table, etc. It is desirable to test when the target in-memory KVS is filled up with KVs, rather than using only a few KVs because these objects increase as more KVs are, and checking whether the error handlers keep the consistency of the objects is mandatory. Feeding tens to hundreds of gigabytes of KVs to the target in-memory KVS in every handlers' test takes quite a long time.

3. MemFI

This paper presents *MemFI* that supports testing memory error handling code in in-memory KVSes. MemFI offers a runtime test environment where the behavior of memory error handlers can be checked. The runtime test is reasonable for the handlers since their behavior depends largely on the internals of the running in-memory KVS. The handlers must recover damaged memory objects cooperatively with the undamaged objects. Some handlers repair damaged objects and modify the undamaged objects so that the recovered in-memory KVSes consistently run. For example, the partial-surgery-aware memcached prunes the damaged KV and updates the chain-based hash to unregister it. Note that the goal of MemFI is different from that of traditional FI tools; traditional FI tools aim to test the reliability of the entire software, while MemFI aims to facilitate the testing of handler behavior during the development phase.

The design of MemFI is driven by the following goals:

- **Triggers memory error handling code:** MemFI forces in-memory KVSes to execute their memory error handlers so that we can check their runtime behavior.
- **Executes various memory error handlers:** MemFI supports the runtime tests of memory error handlers prepared for different memory objects.
- **Quickly restores runtime states of in-memory KVSes:** To efficiently check different handlers on the

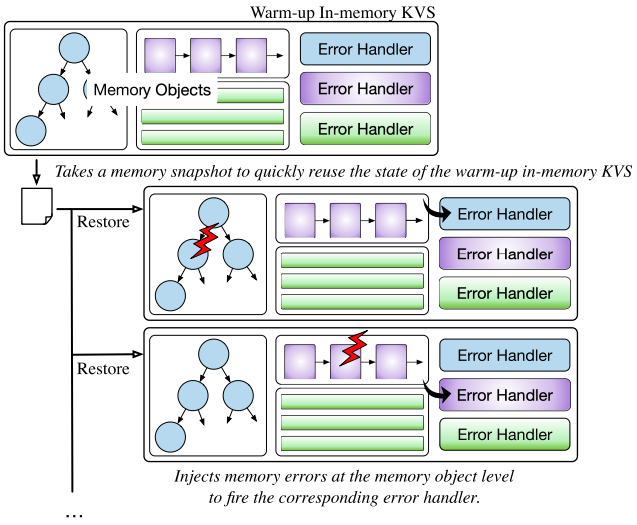


Fig. 1 **MemFI-based Runtime Tests.** *MemFI performs memory error injection at the memory object level to trigger the execution of the target error handler while restoring the running state of the in-memory KVS from a memory snapshot to generate its same state quickly.*

same runtime state of the in-memory KVS, MemFI restores the target memory state without time-consuming command regeneration.

- **Does not modify the in-memory KVS source code:** MemFI is applicable without any modification of the in-memory KVS source code.
- **No hardware modification:** MemFI employs a pure-software approach that requires no modification of the underlying hardware.

To satisfy these design goals, MemFI orchestrates the fault injection and memory checkpointing features. Figure 1 shows an overview of the MemFI-based runtime tests. MemFI performs software memory error injections at the memory object level. Typical fault injectors are random-based; they randomly choose several instructions from all or memory-accessed instructions to flip bits of their operands [28]–[31]. MemFI allows us to intentionally inject the errors into instructions accessing the target objects to trigger the corresponding handler execution. MemFI tracks the internals of the running in-memory, and KVS extracts the virtual addresses of the objects by exploiting its internal knowledge. Also, MemFI restores in-memory KVS running states from memory checkpoints instead of rebuilding the running state from scratch by feeding the same commands. For example, MemFI checkpoints the memory states of the KVS after its warm-up, and modifies the contents on the specified address. We then send a command to access the error-injected address to fire the error handler and check its behavior. To test the next error handler, MemFI restores the KVS's memory states from the checkpoint and inserts the memory errors into the address within the corresponding object.

MemFI helps developers of memory error handlers check their runtime behaviors. Developing the handlers requires deep knowledge of the target application's internals. Testing the memory error handlers involves checks on whether the target handlers successfully repair the damaged object and update the other objects consistently. Also, MemFI targets memory errors in the address space of in-memory KVSes. Their memory error handlers are mainly designed for such memory errors in the heap region since the memory objects on the heap, such as KVs and their metadata, dominate the memory utilization of the in-memory KVSes. Our fault model has two types of memory errors occured at the use-space memory. The first is an n bit-flip error that n bits in the same word are changed. Like traditional software fault injectors, MemFI injects n bit changes into the specified virtual addresses by overwriting their bit sequences. The second is an error that is detectable but not correctable by the ECC hardware, referred to as an ECC-uncorrectable memory error. MemFI mimics the errors' behavior, as described in Sect. 4. MemFI does not target memory error and its handlers inside privileged software such as OS kernels and hypervisors. The user-space memory in in-memory KVSes is much larger than the kernel-space memory. Although memory errors severely impact such privileged software and the mechanisms for memory errors are studied [47], their runtime tests are out of the scope of this paper.

4. Design Details

4.1 Overview

The MemFI runtime test consists of three phases: *warm-up*, *injection*, and *check* phases. Figure 2 shows an overview of these phases. The warm-up phase sets up the target in-memory KVS for the runtime tests of the memory error handlers. In this phase, their developers feed KVs into the in-memory KVS and then take a memory snapshot of it to skip sending KV commands to recreate the same running states. The injection phase is for inserting memory errors at the memory object level. The MemFI engine, running in the address space of the in-memory KVS, extracts the object's virtual addresses and rewrites memory contents on the specified addresses. The check phase sends KV commands to the fault-injected in-memory KVS to trigger the target error handler. After checking its behavior, we restore the in-memory KVS state from the memory snapshot taken in the warm-up phase and move to the next trial to test other memory error handlers.

Designing MemFI poses several technical challenges: (1) How does MemFI know the virtual addresses of the memory objects in the in-memory KVS?, (2) When does MemFI insert memory errors in the target memory object?, (3) How does MemFI mimic ECC-uncorrectable memory errors? MemFI overcomes these challenges with software techniques described as follows.

4.2 Memory Object Identification

Extracting the memory objects in the target in-memory KVS

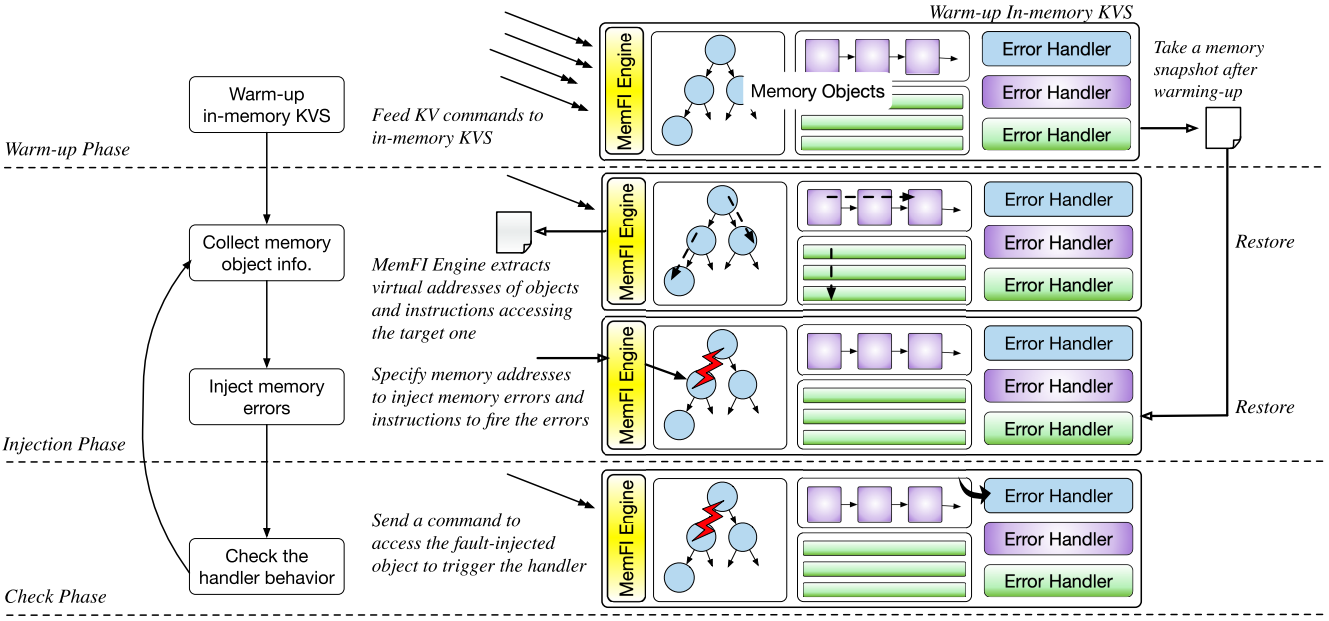


Fig. 2 **Three Phases of MemFI-based Runtime Test.** *The warm-up phase sets up the in-memory KVS by feeding KVs and takes its memory snapshot. The injection phase exposes the virtual addresses of the internal memory objects and specifies the addresses and instructions to fire the memory error, while the check phase checks the runtime behavior of the memory error handlers corresponding to the fault-injected objects.*

and their addresses does not come for free. The data types of memory objects are different even among in-memory KVSes, and the layout of memory objects depends on their running states. Memcached employs a slab allocator and places key, value, and metadata(item) sequentially in slabs. It also manages slab chunks using free and LRU lists. On the other hand, Redis allocates memory using the third-party library and connects KV's metadata(dictEntry) to the corresponding key and value. The number of the objects and the pointer connections between them change dynamically as the in-memory KVSes get and update KVs. To extract the virtual addresses of memory objects, we also need to have application-level knowledge of the memory layout. The knowledge includes object semantics, data types, the addresses of global variables, and relationships between objects. Processes using a process attachment feature like ptrace() and the underlying OS kernel can access the address space of an in-memory KVS. However, these powerful features are insufficient to extract object addresses because the objects cannot be tracked without their types and layouts on the virtual address space.

To address this issue, MemFI employs an application-specific approach that exploits the knowledge of the target in-memory KVS. To obtain the virtual addresses of the target memory object, the MemFI engine, running inside the in-memory KVS, traces memory objects in the address space with the source code level information such as data types, memory object semantics, and global variables. The MemFI engine searches the target object by tracking the running KVS' memory. The engine returns the virtual address of the object and modifies its contents as a memory error.

A typical scenario is as follows. When a developer completes the implementation of a memory error handler, he or she performs MemFI-based runtime tests for the handler. The developer first queries the MemFI engine to the address of the memory object corresponding to the handler. The engine tracks the internals of the in-memory KVS to find the requested memory object and return its address. Then, the developer inserts a memory error to the address using the MemFI engine and sends KV commands so that the fault-injected in-memory KVS can invoke the target handler. Note that MemFI engines must be designed for each in-memory KVS since they leverage application-specific knowledge such as global variables and object types. As case studies, we prototypes MemFI engines for memcached and Redis, as described in detail in Sect. 6 and Sect. 7.

4.3 Error-Firing Control

Although the object-level fault injection efficiently triggers the target error handler, we must consider the timings of accessing the object for a comprehensive runtime test. Access to the same object in the command processing and maintenance tasks in the in-memory KVSes occurs multiple times. For example, memcached accesses the requested KV's metadata 36 times in a get command processing. A naive approach is to insert an error into the target object in the injection phase and start the check phase by sending a command. The approach covers the only case where the in-memory KVS faces the error at the first access to the object in the command processing; we miss the handler checks at the other access points.

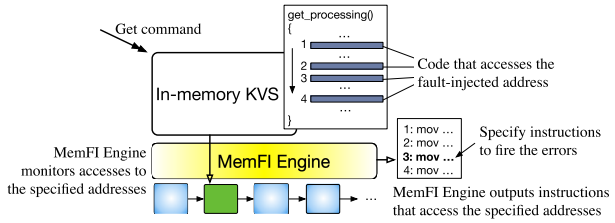


Fig. 3 **Error-firing Control in MemFI** *The MemFI engine traces memory accesses to the given addresses, extracts the accessing instructions, fires the memory error at the chosen instructions on the running in-memory KVS.*

MemFI leverages a dynamic binary instrumentation (DBI) technique to observe the handler's behavior at any point of accessing the damaged memory object. Figure 3 shows an overview of the fault injection point extraction in MemFI. MemFI fires a memory error at the only point specified in advance. In the injection phase, the MemFI engine detects all the instructions accessing a given object in a command processing by monitoring the memory accesses, and we specify the instructions to fire the injected errors. In the check phase, the engine hooks the instructions by a DBI tool at runtime, fires the errors, and executes the hooked instructions, resulting in the execution of the corresponding error handler.

To check a memory error handler in the MemFI-based runtime test, in the injection phase, a developer obtains the virtual address of the target memory object in the warm-up in-memory KVS and passes the MemFI engine the address for the memory error injection. The MemFI engine next detects the injection candidates of the instructions accessing the memory address in processing a given command. The developer specifies a candidate, and then the engine restores the in-memory KVS, whose state is just before processing the command. When the in-memory KVS executes the specified instruction, the MemFI engine hooks the instruction and injects the memory error to the virtual address. The instruction accesses the memory object damaged by the fault injection and triggers the execution of the target handler. If the developer performs a comprehensive test or is hard to specify the error-firing instruction, he or she can resort to injecting errors into the instructions one by one. Conversely, the developer can specify the error-firing instruction when the test scenarios are clear. These manual tasks are non-trivial and sometimes even challenging for developers of the target software. One of our future work is to mitigate these tasks by preparing support tools and/or automating the tasks, as described in Sect. 9.

Note that MemFI's checkpoint-based restoration does not guarantee that the internals of the in-memory KVS is always the same in receiving a command. Some in-memory KVSes, like Memcached, handle KV commands by multiple worker threads that are scheduled non-deterministically. Even if such in-memory KVSes receive the same command at a time, the running state of the in-memory KVS is not always the same. The MemFI-based runtime tests can fail to fire the memory errors since the specified instruction cannot be executed due to the nondeterministic thread execution. MemFI can avoid this problem by employing deterministic thread scheduling mechanisms [48], [49], though we have never faced any problems come from nondeterministic thread scheduling in our experiments.

4.4 ECC-Uncorrectable Memory Error Injection

We also take care of the injection of ECC-uncorrectable memory errors. When a memory region damaged by ECC-uncorrectable memory errors is accessed, the ECC module causes an interrupt with their physical addresses. The OS kernel typically terminates the processes whose address space involves those damaged regions. Specifically, When an in-memory KVS accesses the memory contents damaged by ECC-uncorrectable memory errors, the ECC module causes a non maskable interrupt (NMI). The OS kernel catches it and then invokes the NMI handler. The NMI handler checks the physical address informed by the ECC module and then identifies the accessing process. The default NMI handler terminates the process. In the partial-surgery [15], the extended NMI handler notifies the process of the error and exposes the virtual address of the damaged memory contents. And then, the in-memory KVS invokes the recovery handler of the partial-surgery. Even if multiple-bit errors are injected to the target memory address by rewriting the current contents from software fault injection tools like bit-flip injections, these rewrites are regular updates from the viewpoint of the ECC module; thus, such software-level rewrites cannot cause the hardware behavior in the ECC module error detection.

MemFI mimics the software behavior in accessing the memory address damaged by ECC-uncorrectable memory errors. To do so, MemFI forces the underlying OS to invoke the extended NMI handler. MemFI leverages a kernel-level mechanism and a DBI tool. Specifically, memory addresses to inject an ECC-uncorrectable memory error are specified in the injection phase described in the previous section. The MemFI engine hooks the instruction accessing the address and issues a unique system call whose arguments are the memory address and PID. The OS kernel sends a signal and tells the address to the process with the PID. The signal calls the recovery handler for ECC-uncorrectable memory errors.

5. Implementation

We prototype MemFI on a real-world in-memory KVSes, memcached and Redis. We implement a MemFI engine in memcached 1.6.9 amd Redis 6.2.7 on Linux 5.10.0. These MemFI engines run as a dynamic link library in the same address space. The engines hook __libc_start_main() using LD_PRELOAD and spawns a thread that receives requests for invoking MemFI's mechanisms such as fault injection, object identification, memory access tracing, and checkpointing via network messages with the different port number from the in-memory KVS. In receiving a request to perform a MemFI

mechanism, the engines invoke the requested function and return the result.

The prototypes use Intel Pin [50] to trace and hook the memory accesses. Intel Pin is a dynamic binary instrumentation tool that allows us to hook specific instruction execution and function calls. Our prototypes start hooking memory access instructions in the injection phase and show the instructions that access the specified addresses. The prototypes modify the target memory contents or mimic the ECC module error detection at the specified instructions. The prototypes also use CRIU [51] to checkpoint memory snapshots of the running in-memory KVSes. CRIU is a Linux-based checkpointing mechanism that allows us to take a snapshot of the running processes and restore it. The prototypes checkpoint the running memcached and Redis by spawning a process that executes CRIU commands for memory checkpointing.

6. Case Study: Memcached

To show the effectiveness of MemFI, we check memory error handlers using a MemFI prototype for memcached 1.6.9. We implement two recovery mechanisms for memory errors, software-ECC [17] and the partial-surgery [15], and test their runtime behaviors using MemFI.

6.1 Memcached Internals

Memcached is a multi-threaded in-memory KVS, and its index is based on a hash table. The threads handle requests such as set and get concurrently. When a set command is sent to store a KV, memcached allocates a memory region to save it, calculates a hash value from the key, and registers it in the hash table. Memcached searches the hash table for the target value in the get command.

In memcached, the memory region is managed by its slab allocator. Instead of allocating KV regions in an on-demand manner, the slab allocator creates a slab whose size is 1 MB by default and divides it into chunks. Each slab consists of chunks of a fixed size, but the chunk size varies among slabs. The slabclass groups the slabs whose chunk sizes are the same. To avoid memory fragmentation as much as possible, the slab allocator preserves a value as a chunk having the same size or closest to its size. LRU and free lists chain the chunks in each slabclass. When a value is deleted, the slab allocator chains the chunks of the KV to the free list instead of freeing them.

6.2 MemFI Memcached Engine

The MemFI memcached engine tracks the memcached's memory objects to obtain their addresses. Figure 4 overviews the memory objects of memcached. The MemFI engine obtains the addresses of KVs and memory objects for their management: slabclass and hash table. Our prototype shows the addresses of key, value, and metadata (item). The prototype searches the KV from a given key and returns its addresses.

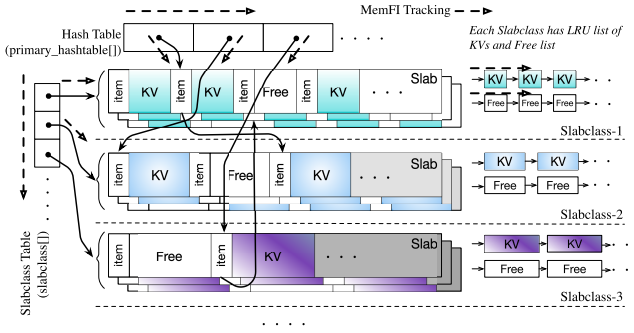


Fig. 4 **Memcached Internals.** *MemFI identifies the virtual addresses of KVs and their metadata (item) by tracking them on hash chains, slabclass pointers, and LRU/Free-lists. MemFI also shows the top addressess and sizes of primary_hashtable[] and slabclass[].*

It also tracks the primary_hashtable[] and the array of slabclass_t as the management objects. When requested, the engine returns their top addresses and sizes.

To help decide which KVs should be fault-injected, the MemFI engine exposes the current states of some management objects. The MemFI engine tracks the hash table and LRU list and shows their chains based on a given parameter. For the hash table, the engine receives an index and shows the chain of its entry in the table while receiving the slabclass number and showing its LRU list chain. The engine also shows all the KVs in a slab by specifying the slabclass number and slab ID. It traces the target slab to extract the KV's information, checking the size of the KV and it_flags, a flag that the KV is valid or not, in item.

The MemFI engine supports checking the handlers' behavior in command processing and maintenance tasks. Memcached periodically updates the LRU lists in slabclasses using a dedicated thread while recreating and rehashing the hash table when the KVs become full. These updates involve accesses to items. Runtime tests must cover such cases since they trigger the error handlers by accessing the damaged memory object. Although some maintenance tasks are done every second and triggered by commands, the memcached never performs the other tasks, like rehashing, until the corresponding event occurs. Our memcached engine forces the memcached to perform such an event-driven task by calling the internal functions in memcached to perform runtime tests smoothly. The prototype does so by waking up the assoc_maintenance_thread performing table rehashing.

6.3 Software-Based ECC

Software-based ECC protects KVs against memory errors. The software-based ECC detects and corrects the damaged KVs by performing the BCH encode [52] for metadata, key, and value regions and comparing their contents with the checksums to detect and fix the KV damaged by memory errors. The BCH-based checks can detect d bit errors and fix t bit errors, where d and t are variables with two or more values. Our prototype of software-based ECC, integrated into memcached, sets $d = 3$, $t = 2$ and calculates the checksums

when a KV is set and updated. The prototype performs the BCH encode for item, key, and value on slabs every update. When accessing them, the prototype recalculates the checksums from the current contents and compares them with the preserved ones. If necessary, the prototype fixes the contents using the BCH-based correction.

To confirm that our software-based ECC mechanism protects target memory objects against memory errors, we perform runtime tests of the prototype using the MemFI memcached engine. We set up memcached to be filled with 10 million KVs, each 5 to 10 KB in size. Its memory utilization is 10 GB. We insert bit-flip errors into metadata, key, and value at various access points using MemFI. We insert a single bit-flip into each object at the access points in the code path of the get command and check whether our software-based ECC mechanism repairs the object contents.

The result reveals that the MemFI engine successfully inserts the bit-flip errors into the target object and timing, and the prototype of software-based ECC repairs the object contents consistently with other management objects. Here, we show the two cases. One is that the get command accesses the fault-injected address in the key area. The other is that the LRU-list maintenance task, lru_maintainer touches the fault-injected region in the item area. In both cases, our software-based ECC detects and corrects the injected errors.

We also have to warm-up the memcached many times since it consists of various KVs and their metadata and frequently accesses them in the command processing and maintenance tasks. After a runtime check is done, we need to warm-up memcached again. The snapshot restoration of 10 GB memcached used in this experiment takes 20 seconds, while the time to warm-up memcached by feeding the KVs is 106 seconds. MemFI significantly shortens the time for testing the memory error handling.

6.4 Partial-Surgery

The partial-surgery is an approach to enforcing in-memory KVSes to survive ECC-uncorrectable memory errors. The approach finds memory objects in the error-affected memory pages and reorganizes the in-memory structures when the ECC hardware module detects ECC-uncorrectable memory errors. Our prototype of partial-surgery aware memcached handles ECC-uncorrectable memory errors in slabs. The prototype releases the damaged slab and deletes the KVs in that slab from the hash table and LRU/Free lists. If the memory pages where the other memory objects reside are corrupted, It terminates in a fail-stop manner. The prototype updates the slabclass metadata when discarding a corrupted slab, and the Linux kernel never reuses the damaged pages. It eliminates the pointer to the corrupted slab and overwrites the number of slabs. Also, our prototype accesses the valid KVs by directly accessing the undamaged slabs, and re-registers them in the hash table to reconstruct its chains and LRU list. In so doing, it also extracts the free regions of the slabs and chains them in each slabclass to restore the free list.

We test the recovery behavior of our partial-surgery aware memcached with MemFI. In the runtime tests, MemFI inserts ECC-uncorrectable memory errors into various objects since the prototype's memory error handlers depend on the memory object. MemFI mimics the ECC-uncorrectable memory error by sending the signal to the process in accessing the target address. Our runtime test inserts the errors into slabs and other objects like a hash table. The injections are done at all the accesses to the damaged objects in get and set paths.

The prototype of the partial surgery consists of four types of memory error handlers: hashtable, key, value, and items. Memcached performs one access to the hashtable in processing a get command. Similar to the software-based ECC case, it can take quite a long time with existing fault injectors and warm-up from scratch since checking the handlers requires the injection of memory errors into the target memory addresses at the specific instructions and repeatedly warm-up of memcached. The MemFI-based runtime tests perform them efficiently, as described above.

7. Case Study: Redis

We also prototype MemFI and implement two mechanisms for memory errors on Redis 6.2.7. We implement the partial-surgery and the memory error aware KV saving, borrowing the idea of software-based ECC.

7.1 Redis Internals

Redis is a widely-used KVS whose memory management differs from memcached's. The big difference is that Redis's memory management relies on third-party libraries and allocates a memory chunk on demand. Redis allocates a memory chunk for metadata, key, and value for every set request, and the chunks point to each other. Redis frees the corresponding chunks via the library functions when a delete request arrives. Also, it maintains a chain hash table as the KV index, similar to memcached.

When a KV set request arrives, Redis first creates KV metadata, called dictEntry, and registers it to a dictht, a chain-based hash table used as the index. The chains consist of dictEntries. Then, Redis allocates the key and value buffers and links them to the dictEntry. In so doing, Redis calls library functions to allocate memory regions for the objects. There are robjs between KVs and dictEntry. Redis also looks up the hash table with the requested key and returns the value to the clients upon receiving get requests. Redis has another dictht to rehash the hash table incrementally.

7.2 MemFI Redis Engine

The MemFI Redis engine obtains the current memory object layout by tracking the hash table. Figure 5 shows an overview of Redis's memory objects. The MemFI engine detects the addresses of KVs, their metadata (dictEntry and robj), and the hash table. The engine searches a KV from a given key

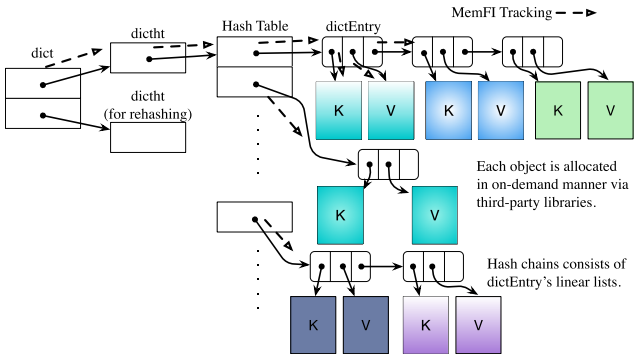


Fig. 5 **Redis Internals.** *MemFI identifies the virtual addresses of KVs and their metadata (dictEntry) by tracking them on hash chains. MemFI also shows the top address and size of dictht[].*

and returns the addresses of its key, value, dictEntry, and robj. Also, the engine returns the top address and size of dictht[] if requested.

To help know the internals of the running Redis, the MemFI Redis engine walks the hash chain of a given index in the hash table and shows the addresses of the KVs' objects in the chain. The Redis engine is much simpler than the memcached one due to Redis' simple design. Redis does not manage LRU- and free-lists and does not employ its memory allocator and background threads that perform maintenance tasks.

7.3 Partial-Surgery

We elaborate the partial-surgery approach into Redis 6.2.7. Our Redis employs a memory allocator specialized for KVs and dictEntries. The memory allocator pre-allocates a memory pool exclusively used to avoid interleaving the library's metadata in their memory pages. Its design is based on the slab allocator that creates a slab pool in the initialization, divides it into chunks, and allocates them. The prototype prepares slabs and slabclasses for each type of memory object. For example, it allocates a key from the chunks of a slab for the key. To do so, the prototype annotates the memory allocation with the type of memory object so that our memory allocator can identify the slabs to use. It has three groups of slabs: key, value, and dictEntry. This design releases us to restore the library's metadata since the library metadata is in the header of the memory pool. Although the memory allocator of Redis is dramatically changed from the original one, the MemFI engine can trace the KVs in the same way since any entries of dictEntry and robj are not modified.

To confirm that our Redis protects target memory objects against ECC-uncorrectable memory errors, we perform runtime tests of the prototype using the MemFI Redis engine. We set up Redis to be filled up with 10 million KVs, each of which is 5 to 10 KB in size, a total of 10 GB, and insert ECC-uncorrectable memory errors into dictEntry, key, and value. We insert the errors into each object at all the access points in the code path of the get and set commands and check whether the partial-surgery handlers repair the faulted objects. We insert the errors into each object at all the access points in the code path of the get and set commands and check whether the partial-surgery handlers repair the faulted objects.

The result reveals that MemFI allows us to check the runtime behavior of our partial-surgery aware Redis against ECC-uncorrectable memory errors. For example, when MemFI inserts an ECC-uncorrectable memory error into a dictEntry, the Redis identifies the address of the damaged dictEntry and prunes it and the successors of its chain. It also unregisters the KV from the hashtable when a key or value is damaged by an ECC-uncorrectable memory error of MemFI.

The snapshot restoration of 10 GB Redis used in this experiment takes 24 seconds, while the time to warm-up Redis by feeding the KVs is 415 seconds. As in the memcached case, MemFI significantly shortens the time for testing the memory error handling in Redis.

7.4 Memory Error-Aware KV Saving

The KV saving feature in Redis stores the current in-memory KVs on storage. When receiving a save request, Redis spawns a child process to store the KVs and concurrently handle KV commands. If the KVs are damaged and their contents are stored in storage, Redis restored from the KV log returns the wrong values of the KVs. We extend the KV saving mechanism to be memory error aware, borrowing the idea of software-based ECC. Our KV saving checks whether or not the KVs are damaged by memory errors in storing them to storage. As the original software-based ECC mechanism described in Sect. 6.3, the extended Redis performs the BCH encode for a KV and its metadata and preserves the checksums when they are set and updated. The memory error-ware KV saving calculates the checksums of a KV and its dictEntry with the BCH code and compares the checksums to the last ones before storing. The KV saving writes the KVs' contents into the log if the current checksums are equal to the last ones. If not equal, our mechanism repairs the KV contents or disposes of the KV if the contents restoration using BCH is impossible.

Similar to other case studies, we perform a MemFI-based test to check the runtime behavior of the memory error-aware KV saving. We use Redis with the same configuration as in the previous section and perform our KV saving after inserting two- and three-bit errors into various keys, values, and dictEntries. The result shows that the memory error-aware KV saving successfully deals with the KVs damaged by MemFI's injected errors. Our KV saving repairs the contents of KVs and dictEntries with the two bits errors and stores their correct contents. On the other hand, it skips the three bits error-ed KVs to store since the BHC-based restoration cannot fix their contents. Like the previous case, MemFI can perform the runtime test more efficiently than the legacy techniques in our KV saving since we can check the target handler's behavior in a comprehensive manner by injecting and firing bit flips to the appropriate object at the

desirable points with the fast restoration for every trial.

Note that we did not implement the regular software-based ECC on Redis, like the case study of memcached. This is because we avoid the overlap with our memory error-aware KV saving that uses the software-based ECC checking.

8. Compairison with other FI Tools

We compare MemFI with existing fault injectors from the efficiency viewpoint for runtime tests. To do so, we build a trial model to estimate how many FI trials are needed for testing target memory error handlers. Based on the model, we compare MemFI with two types of fault injectors widely used in previous studies: one is to invert the contents of the operand register in a randomly selected instruction from all instructions [29], [53] (*all-random*), and the other is to change the register contents in randomly selected one instruction from all the memory access instructions [54] (*mem-random*).

8.1 FI Trial Model

The time for the FI experiments can be expressed as the "Each Trial Time(ETT)" × "Trial Numbers(TN)". ETT is the time for each fault injection trial. The time-consuming task in testing memory error handlers is to check the consistency of all the memory objects after their recovery. This task has to be done manually with existing automated fault injectors and MemFI, and is dominant of ETT. We note that TN execution time of random-based FIs is smaller than MemFI due to the involvement of its manual tasks like address/instruction specification.

Although the current MemFI-based test involves some manual tasks, including runtime information checks and error injection specification, the trial execution time of random-based FIs is shorter than MemFI. We note that the dominant task in FI-based runtime tests is to check whether the triggered error handler behaves correctly; we manually check the handler correctly detects/fixes the errored objects and whether other objects in the in-memory KVS are consistent. These tasks are required with random-based FIs and MemFI. Thus, we focus on TN, the number of FI trials to generate the desired memory error situation with a fault injector. MemFI's TN is one because it inserts a bit-flip into the target virtual addresses and fires it at a time specified in advance. On the other hand, the TN of the existing fault injectors is based on a binomial distribution; the injectors randomly insert errors to the target instruction (success) or not (fail). This is widely used in the dependable computing communities [55]. The binomial distribution is a probability distribution that the number of successes X follows in n Bernoulli trials. If the probability of success per trial is p, the probability of success k times within n trials, i.e., $P(X = k)$ can be calculated from the equation: $P(X = k) = {}_nC_k p^k (1-p)^{n-k} (k = 0, 1, 2, \ldots, n)$.

We set the success probability p as the probability that a random-based fault injector hits the target instruction where we want to fire the error. This is expressed by: $p = \frac{Number\ of\ Target\ Instructions\ (I\_target)}{Number\ of\ Total\ Instructions\ (I\_total)}$. I_total is the number of the instructions into which the fault injector inserts bit flips as the candidates, while I_target is the number of instructions to be fired for generating the desired memory error condition. By taking the sum of $P(X = k)$ from $k = 1$ to n, we can determine the probability of at least one hit in n trials. For example, an event with $p = 0.01\%$ has a probability of about 63% for $n = 10000$ and a probability of about 95% for $n = 30000$ to hit at least once. We define the TN in the random FI approaches as n when the probability of at least one hit exceeds 95% of all trials, as follows: $TN = n \left(\sum_{k=1}^{n} P(X = k) >= 0.95\right)$.

8.2 Comparison Using Runtime Information

We estimate the number of trials (TN) for each injector on the memcached and Redis cases described in Sect. 6 and Sect. 7. In the case study, we used a workload that sets 10 million items of different sizes from 10 B to 5 KB (10 GB in total) and then sends a get request for each item.

In memcached case, to check the memory error handlers, we have to generate nine error situations; the target objects are three (item, key, and value), and events of accessing them are three(set, get, lru_crawler_thread). We calculate the TNs of the all-random and mem-random for each error case based on the above model. To determine the p, we collect various runtime information, shown in Table 1, using Intel Pin. We estimate the trial costs using the collected values. For example, I_target on the item error case in a get command is $36 \times 10{,}000{,}000 = 360{,}000{,}000$ times. Therefore, the probability p using all-random is $360{,}000{,}000/171{,}405{,}295{,}000 = 0.21\%$. Table 3 summarizes the TN for each error case. The result reveals that the random-based injectors require hundreds to tens of thousands of FI trials. TN, even with mem-random, exceeds 10,000 in some cases. Since MemFI can always inject errors into the target instruction at the desired time, its TN is one.

The Redis case is similar to the memcached one. Six error situations are required in the Redis case; the target objects are three (dictEntry-robj, key, and value), and the events of accessing them are three(set, get). As in the memcached case, we calculate the TNs of the all-random and mem-random for each error case using the trial model and runtime information shown in Table 2. Table 4 summarizes the TN for each error case. The table shows that the existing random FI methods require hundreds to tens of thousands of trials.

For both KVSs, TN of existing random methods was found to be enormous, while MemFI's TN is one due to its memory object-level injection and error firing control. In addition, MemFI restores target KVS internals by leveraging a memory snapshot and allows us to perform FI trials smoothly. In our experimental setting, as described in Sect. 6 and Sect. 7, the restoration time is reduced by 82% for the memcached case and 94% for the Redis case.

Table 1 Instructions under Memcached Workload.

All instructions		171,405,295,000
Memory-Access instructions		57,875,621,000
set command for 1 KV	item	84
	key	6
	value	1
get command for 1 KV	item	36
	key	6
	value	1
lru_crawler thread for 1 KV	item	21
	key	2
	value	1

Table 2 Instructions under Redis Workload.

All instructions		124,884,126,000
Memory-Access instructions		49,045,398,000
set command for 1 KV	dictEntry+robj	22
	key	13
	value	4
get command for 1 KV	dictEntry+robj	11
	key	2
	value	3

Table 3 Trial Numbers in Memcached Experiments.

Objects / Operation	All-random	Mem-random	MemFI
item / set	610	200	1
key / set	8,560	3,000	1
value / set	51,650	17,650	1
item / get	1,430	485	1
key / get	8,560	3,000	1
value / get	51,650	17,650	1
item / crawler	2,495	835	1
key / crawler	27,240	8,800	1
value / crawler	51,650	17,650	1

Table 4 Trial Numbers in Redis Experiments.

Objects / Operation	All-random	Mem-random	MemFI
dictEntry+robj / set	1,670	680	1
key / set	3,000	1,100	1
value / set	9,370	3,700	1
dictEntry+robj / get	3,370	1,365	1
key / get	18,730	7,490	1
value / get	12,500	4,910	1

9. Discussions and Limitations

Memory errors severely impact modern in-memory KVSes, and software mechanisms for memory errors are essential to enhance their dependability. MemFI facilitates the runtime test of these mechanisms by identifying the memory objects, injecting errors to the addresses within the target object, and firing them at the various access points. It has several discussions and limitations to be addressed in future work.

Injection of Memory Errors to Heap Objects: In-memory KVSes, most of the memory consumption is occupied by objects in the heap area. For example, in Memcached, the memory usage of slab accounted for 99.4% of the total out of 16 GB memory and in Redis, KVs and dictEntries accounted for 94.5% of the total. MemFI performs more effective runtime tests by covering many types of memory objects, such as text regions and global variables.

Other Types of In-memory Applications: The target of MemFI in this paper focuses on in-memory KVSes, a widely-known in-memory application. Like in-memory KVSes, other types of in-memory applications such as machine learning and graph analysis also face memory errors relatively to legacy stateless applications due to their large memory footprints. Although we have to design and implement MemFI engines to the target software, the concept of MemFI, which injects memory errors at the memory object level and quickly restores the runtime states of the target one, is applicable and effective to other in-memory applications. As the first step in investigating ways to efficiently test memory error handlers for in-memory applications, this work focuses on the typical one, namely in-memory KVSes. Ways to facilitate the design of an application-specific MemFI engine need to be explored to make MemFI more applicable and showing how effective MemFI-based runtime tests are to in-memory applications is one of our future work.

Automatic Object Layout Analysis: The types of memory objects depend on applications. The prototypes of the MemFI engines for memcached and Redis traces the target objects *manually*: We select the target memory objects such as slabs and dictEntry, and implement the object trace functionalities with the application-specific knowledge. To facilitate the implementation of MemFI engines, runtime tools automatically extract the internals at the memory object level, and the addresses of the objects are valuable.

Automatic Injection Scenario Generation: The prototypes determine the memory addresses and timing for the memory error injection in an interactive manner: we manually specify the addresses and timing based on the object layouts and memory access traces extracted by the MemFI engines. These tasks are harder as memory error handlers, memory objects, and access points are more. Exploring automatic testing ways that generate memory error injection scenarios, inject memory errors based on the scenarios, and check behaviors of the error handlers is an important topic. By using automatic scenario generation and object analysis described above, we can perform a memory error injection campaign to various memory error handlers for behavior checks and comparison of different types of handlers.

10. Related Work

Compared to the state-of-the-art techniques, the representative feature of MemFI is to inject memory errors at the memory object-level at runtime. This allows us to trigger the execution of the target error handlers, leading to the efficient checks of their behavior.

Software Fault Injectors (SFIs) are effective in observing the runtime behaviors of the target software against various faults. Typical SFIs mimic a memory error by inserting bit-flips into the destination register in a randomly chosen instruction [28]–[31]. The approach is inefficient in testing memory error handlers. Since the handler's behavior depends on the memory object, a developer has to repeatedly perform fault injection until the injector inserts the bit-flip to the memory object whose damage triggers the target memory error handlers. Yim et al. present an SFI technique that tracks the memory accesses of the target applications at the

OS kernel level, extracts accessed memory pages, and injects errors into them [54]. Since it is application-independent but cannot insert errors into the target memory objects, we cannot trigger the corresponding memory error handler intentionally. Other SFIs [24], [56] are used for checking error handlers by hooking library functions to return error values. Such library function errors cannot lead to the execution of memory error handlers triggered by accessing the damaged addresses in their target objects.

Due to the unique characteristics of the memory error handlers in in-memory KVSes, as described above, existing dynamic/static analysis methods take much work to test their code. Fuzzing is a runtime testing method to detect bugs and discover vulnerabilities. It generates inputs for the target applications to walk as much of their program code as possible. Its approaches generate inputs from using the specific format or grammar [25], [57], [58], create mutation sets by changing the seeds [26], [27], [59]–[64], and uses both methods [65]–[69]. Although these approaches improve code coverage, it is hard to test memory error handlers; since these handlers are triggered only when the target applications access the memory address damaged by a memory error, feeding KV commands cannot fire the error handlers.

FIFUZZ [24] is a hybrid approach of fuzzing and fault injection. To improve the code coverage, FIFUZZ feeds various inputs to the target applications while hooking library calls to inject errors so that error handling code can be executed. Both techniques cannot mimic ECC-uncorrectable memory error, failing to trigger their handlers. TTVM [70] and FaultVisor [71], [72] leverage system virtualization to debug operating system components by generating hardware events. These approaches do not insert memory errors.

Dynamic analysis using symbol execution is helpful to check the runtime behavior of the target applications [73], [74]. It substitutes values into variables used in the conditional statements to execute as many branches as possible. The symbol execution cannot trigger handlers for ECC-uncorrectable memory errors.

Static analysis is an effective approach to detecting software bugs. It analyzes the source code of the target applications statically and checks the code, even error handling code, by using pattern matching, data/control flow graphs, and so on [75], [76]. However, the static analysis is hard to check the memory error handlers since these behaviors depend on the internals of the in-memory KVSes at runtime.

11. Conclusion

Since memory errors severely impact in-memory KVSes, software mechanisms for hardening them against memory errors have been explored. Testing the error-handling code of these software mechanisms is challenging due to its unique characteristics. This paper presents MemFI that supports efficient runtime tests for memory error handlers of in-memory KVSes. MemFI injects memory errors at the memory object level and fires them at various instruction points while quickly warming up the target in-memory KVSes by leveraging a memory checkpointing feature. We prototyped MemFI on two real-world in-memory KVSes, memcached and Redis, and performed MemFI-based runtime tests for memory error handlers, such as the partial-surgery and software-based ECC, integrated into them. The experiments with the prototypes reveal that the MemFI-based runtime test allows us to efficiently check the error-handlers.

Acknowledgments

This work was supported in part by JST, PRESTO Grant Number JPMJPR21P9, Japan.

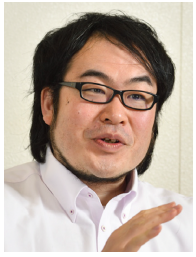
References

[1] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: A large-scale field study," Proc. Eleventh International Joint Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '09), pp.193–204, 2009.
[2] X. Li, M.C. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," Proc. 2010 USENIX Conf. USENIX Annual Technical Conference (USENIX ATC '10), pp.6–6, 2010.
[3] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field," Proc. 2015 45th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN '15), DSN '15, pp.415–426, 2015.
[4] L. Zhang, B. Neely, D. Franklin, D. Strukov, Y. Xie, and F.T. Chong, "Mellow writes: Extending lifetime in resistive memories through selective slow write backs," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '16), pp.519–531, 2016.
[5] M. Zhang, L. Zhang, L. Jiang, Z. Liu, and F.T. Chong, "Balancing performance and lifetime of mlc pcm by using a region retention monitor," Proc. IEEE International Symposium on High Performance Computer Architecture (HPCA '17), pp.385–396, 2017.
[6] C.L. Chen and M.Y. Hsiao, "Error-correcting codes for semiconductor memory applications: A state-of-the-art review," IBM J. Res. Dev., vol.28, no.2, pp.124–134, 1984.
[7] M.Y. Hsiao, "A class of optimal minimum odd-weight-column secded codes," IBM J. Res. Dev., vol.14, no.4, pp.395–401, 1970.
[8] M.B. Sullivan, N. Saxena, M. O'Connor, D. Lee, P. Racunas, S. Hukerikar, T. Tsai, S.K.S. Hari, and S.W. Keckler, "Characterizing and mitigating soft errors in gpu dram," MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), pp.641–653, Association for Computing Machinery, 2021.
[9] M. Patel, J.S. Kim, H. Hassan, and O. Mutlu, "Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices," 2019 49th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN '19), pp.13–25, 2019.
[10] X. Du and C. Li, "Predicting uncorrectable memory errors from the correctable error history: No free predictors in the field," Proc. International Symposium on Memory Systems (MEMSYS '21), Association for Computing Machinery, 2022.
[11] J. Yang, Y. Yue, and K. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at Twitter," Proc. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20), pp.191–208, 2020.
[12] B. Berg, D.S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G.R. Ganger, "The CacheLib Caching Engine: Design and Experiences at Scale," 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20), pp.753–768, 2020.

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," Proc. 12th ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '12), pp.53–64, 2012.
[14] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H.C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," Proc. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13), pp.385–398, 2013.
[15] T. Shimomura and H. Yamada, "Hardening In-memory Key-value Stores against ECC-uncorrectable Memory Errors," Proc. 52nd Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN '22), pp.509–521, 2022.
[16] C. Borchert, H. Schirmeier, and O. Spinczyk, "Compiler-Implemented Differential Checksums: Effective Detection and Correction of Transient and Permanent Memory Errors," Proc. 53rd Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN '23), pp.81–94, 2023.
[17] Y. Li, H. Wang, X. Zhao, H. Sun, and T. Zhang, "Applying Software-based Memory Error Correction for In-Memory Key-Value Store: Case Studies on Memcached and RAMCloud," Proc. Second International Symposium on Memory Systems (MemSys '16), pp.268–278, 2016.
[18] K. Taranov, G. Alonso, and T. Hoefler, "Fast and strongly-consistent per-item resilience in key-value stores," Proc. Thirteenth EuroSys Conference (EuroSys '18), pp.39:1–39:14, 2018.
[19] "memcached-a distributed memory object caching system."
[20] "Redis," Accessed: 2024-01-03. https://redis.io/.
[21] "Amazon Elastic Compute Cloud," Accessed: 2024-01-03. http://aws.amazon.com/ec2/.
[22] "Google Compute Engine," Accessed: 2024-01-03. https://cloud.google.com/why-google-cloud.
[23] A. Goel, B. Chopra, C. Gerea, D. Mátáni, J. Metzler, F. Ul Haq, and J. Wiener, "Fast Database Restarts at Facebook," Proc. 2014 ACM SIGMOD international conference on Management of data (SIGMOD '14), pp.541–549, 2014.
[24] Z.M. Jiang, J.J. Bai, K. Lu, and S.M. Hu, "Fuzzing error handling code using Context-Sensitive software fault injection," 29th USENIX Security Symposium (USENIX Security '20), pp.2595–2612, 2020.
[25] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," 26th Annual Network and Distributed System Security Symposium(NDSS '19), 2019.
[26] M. Zalewski, "American fuzzy lop." https://lcamtuf.coredump.cx/afl/.
[27] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," 26th Annual Network and Distributed System Security Symposium(NDSS '19), 2019.
[28] A. Thomas and K. Pattabiraman, "Llfi : An intermediate code level fault injector for soft computing applications," 2013.
[29] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," 2014 44th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN '14), pp.375–382, 2014.
[30] L. Palazzi, G. Li, B. Fang, and K. Pattabiraman, "A tale of two injectors: End-to-end comparison of ir-level and assembly-level fault injection," 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE '19), pp.151–162, 2019.
[31] T. Tsai, S.K.S. Hari, M. Sullivan, O. Villa, and S.W. Keckler, "Nvbitfi: Dynamic fault injection for gpus," 2021 51st Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN '21), pp.284–291, 2021.
[32] G. Georgakoudis, I. Laguna, H. Vandierendonck, D.S. Nikolopoulos, and M. Schulz, "Safire: Scalable and accurate fault injection for parallel multithreaded applications," 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS '19), pp.890–899, 2019.
[33] N. Nezu and H. Yamada, "Supports for Testing Memory Error Handling Code of In-memory Key Value Stores," Proc. 19th European Dependable Computing Conference (EDCC '24), pp.41–48, 2024.
[34] Y. Kim, R. Daly, J. Kim, C. Fallin, J.H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA '14), pp.361–372, 2014.
[35] S. Levy, K.B. Ferreira, N. DeBardeleben, T. Siddiqua, V. Sridharan, and E. Baseman, "Lessons learned from memory errors observed over the lifetime of cielo," Proc. Int. Conf. High Performance Computing, Networking, Storage, and Analysis (SC '18), pp.43:1–43:12, 2018.
[36] P. Nikolaou, Y. Sazeides, L. Ndreu, and M. Kleanthous, "Modeling the implications of dram failures and protection techniques on datacenter tco," 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '15), pp.572–584, 2015.
[37] B. Sangchoolie, K. Pattabiraman, and J. Karlsson, "One bit is (not) enough: An empirical study of the impact of single and multiple bit-flip errors," Proc. 2017 47th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN '17), pp.97–108, 2017.
[38] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. Lie, D.D. Mannaru, A. Riska, and D. Milojicic, "Susceptibility of commodity systems and software to memory soft errors," IEEE Trans. Computers, vol.53, no.12, pp.1557–1568, 2004.
[39] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generative software-based memory error detection and correction for operating system data structures," 2013 43rd Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN '13), pp.1–12, 2013.
[40] Dong Tang, P. Carruthers, Z. Totari, and M.W. Shapiro, "Assessment of the effect of memory page retirement on system ras against hardware faults," Int. Conf. Dependable Systems and Networks (DSN'06), pp.365–370, 2006.
[41] T.J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," IBM Microelectronics division, vol.11, pp.1–23, 1997.
[42] V. Sridharan, N. DeBardeleben, S. Blanchard, K.B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory errors in modern systems: The good, the bad, and the ugly," Proc. Twentieth Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '15), pp.297–310, 2015.
[43] R.A. Ashraf, R. Gioiosa, G. Kestor, R.F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in hpc applications," Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC '15), pp.1–12, 2015.
[44] S.K.S. Hari, S.V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults," Proc. 17th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '12), pp.123–134, 2012.
[45] M. Didehban, A. Shrivastava, and S.R.D. Lokam, "Nemesis: A software approach for computing in presence of soft errors," Proc. 2017 IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD '17), pp.297–304, 2017.
[46] X. Ni and L.V. Kalé, "Flipback: automatic targeted protection against silent data corruption," Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC '16), pp.335–346, 2016.
[47] T. Iguchi and H. Yamada, "Graceful ecc-uncorrectable error handling in the operating system kernel," Proc. IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE '22), pp.109–120, 2022.
[48] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: Efficient Deterministic Multithreading in Software," Proc. 14th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '09), pp.97–108, 2009.

[49] T. Liu, C. Curtsinger, and E.D. Berger, "Dthreads: Efficient Deterministic Multithreading," Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP '11), pp.327–336, 2011.
[50] "Intel Pin," Accessed: 2024-01-03. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.
[51] "CRIU: Checkpoint/Restore In Userspace," Accessed: 2024-01-03. https://criu.org/Main_Page.
[52] D.S. Lin, Error Control Coding, second ed., Fundamentals and Applications, Prentice Hall, 2004.
[53] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "Llfi: An intermediate code-level fault injection tool for hardware faults," Proc. 2015 IEEE Int. Conf. Software Quality, Reliability and Security Companion (QRS-C '15), pp.11–16, 2015.
[54] K.S. Yim, Z. Kalbarczyk, and R.K. Iyer, "Measurement-based analysis of fault and error sensitivities of dynamic memory," 2010 IEEE/IFIP Int. Conf. Dependable Systems & Networks (DSN '10), pp.431–436, 2010.
[55] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," IEEE Trans. Software Engineering, vol.16, no.2, pp.166–182, 1990.
[56] P.D. Marinescu and G. Candea, "Lfi: A practical and general library-level fault injector," 2009 IEEE/IFIP Int. Conf. Dependable Systems and Networks (DSN '09), pp.379–388, 2009.
[57] P. Godefroid, A. Kiezun, and M.Y. Levin, "Grammar-based whitebox fuzzing," Proc. 29th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '08), pp.206–215, 2008.
[58] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," 2017 IEEE Symposium on Security and Privacy (SP '17), pp.579–594, 2017.
[59] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," Proc. 2016 ACM SIGSAC Conf. Computer and Communications Security (CCS '16), pp.1032–1043, 2016.
[60] A.D. Householder and J.M. Foote, "Probability-based parameter selection for black-box fuzz testing," 2012.
[61] K. Serebryany, "Continuous fuzzing with libfuzzer and address-sanitizer," 2016 IEEE Cybersecurity Development (SecDev '16), pp.157–157, 2016.
[62] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," 2018 IEEE Symposium on Security and Privacy (SP '18), pp.679–696, 2018.
[63] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," Proc. 33rd ACM/IEEE Int. Conf. Automated Software Engineering (ASE '18), pp.475–485, 2018.
[64] V.-T. Pham, M. Böhme, A.E. Santosa, A.R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," IEEE Trans. Software Engineering, vol.47, no.09, pp.1980–1997, 2021.
[65] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," Proc. 21st USENIX Conf. Security Symposium (USENIX Security '12), p.38, 2012.
[66] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," Proc. 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), pp.329–340, 2019.
[67] "PeachFuzzer," Accessed: 2024-01-03. https://peachtech.gitlab.io/peach-fuzzer-community/.
[68] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," 24th Annual Network and Distributed System Security Symposium(NDSS '17), 2017.
[69] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," 2019 IEEE/ACM 41st Int. Conf. Software Engineering (ICSE '19), pp.724–735, 2019.
[70] S.T. King, G.W. Dunlap, and P.M. Chen, "Debugging Operating Systems with Time-Traveling Virtual Machines," Proc. 2005 USENIX Annual Technical Conference (USENIX ATC '05), pp.1–15, 2005.
[71] S. Takekoshi, T. Shinagawa, and K. Kato, "Testing device drivers against hardware failures in real environments," Proc. 31st Annual ACM Symposium on Applied Computing (SAC '16), pp.1858–1864, 2016.
[72] M. Misono, M. Ogino, T. Fukai, and T. Shinagawa, "Faultvisor2: Testing hypervisor device drivers against real hardware failures," Proc. 2018 IEEE Int. Conf. Cloud Computing Technology and Science (CloudCom '18), pp.204–211, 2018.
[73] J.C. King, "Symbolic execution and program testing," Commun. ACM, vol.19, no.7, pp.385–394, 1976.
[74] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A software testing service," SIGOPS Oper. Syst. Rev., vol.43, no.4, pp.5–10, 2010.
[75] H.S. Gunawi, C. Rubio-González, A.C. Arpaci-Dusseau, R.H. Arpaci-Dussea, and B. Liblit, "Eio: Error handling is occasionally correct," Proc. 6th USENIX Conf. File and Storage Technologies (FAST '08), pp.207–222, 2008.
[76] S. Jana, Y. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," Proc. 25th USENIX Conf. Security Symposium (USENIX Security '16), pp.345–362, 2016.

Naoya Nezu received the B.E. degree from Tokyo University of Agriculture and Technology in 2021, and is currently a master student. His interests are in operating systems, dependable systems, and cloud computing.



Hiroshi Yamada received his B.E. and M.E. degrees from the University of Electro-communications in 2004 and 2006, respectively. He received his Ph.D. degree from Keio University in 2009. He is currently an associate professor in the Division of Advanced Information Technology and Computer Science at Tokyo University of Agriculture and Technology. His research interests include operating systems, dependable systems, virtualization, and cloud computing. He is a member of ACM, USENIX and IEEE/CS.