Anatomy of a Bug

John H. Kelm

Thu 2nd Apr, 2009

A bug was found recently in the Rigel tool chain when one particular benchmark running on RigelSim began crashing during nightly runs. RigelSim is a C++ timing simulator that is part of a larger collection of software tools, including a compiler, libraries, and benchmarks, used for modeling the performance of the Rigel accelerator design. The large number of pieces

The benchmark triggering the bug was GJK collision detection. The bug took nearly three hours to reach on a fast workstation, forcing us to think carefully about what techniques to apply to isolate and remove the bug. What follows is a description of the insertion, detection, and removal of a code bug that took nearly four days to find, but less than two hours to correct.

1 Introduction

GJK recently underwent modification to reduce the time spent in the initialization phase of the benchmark. The initialization phase was approaching eight hours on machines used to do nightly runs. An anomaly surfaced quickly: different simulated hardware configurations were showing vastly different numbers of work units being completed in the benchmark. The benchmark code runs differently on each configuration, but the number of work units, or tasks, should be fixed. We believed that there was a bug in the benchmark code.

The maintainer of the benchmark rewrote parts of the code to address the issue. It was not clear that there were any errors in GJK, but every other benchmark was running properly and only GJK had undergone revisions prior to finding the anomaly. The new version of GJK was run only to find that every job, regardless of simulation configuration used, had failed overnight. More importantly, every job failed in the same way.

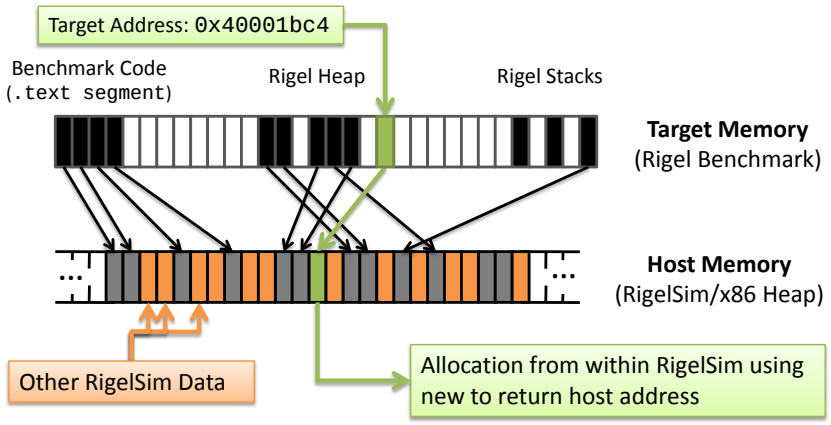

Figure 1: Mapping between target addresses and the host (RigelSim) memory.

2 Bug Discovery and Background

2.1 Continual Regression Testing

In the description of the bug, we will use *target* to refer to state associated with the system being simulated, the Rigel accelerator in this case, and *host* to refer to the system running the simulator, an x86 workstation here. Just like a real machine, the virtual Rigel being simulated has memory. The target memory is mapped to data structures on the host side that will be located on the host's heap. Figure 1 shows the mapping of a target address into the host address space of the RigelSim process. Each block in the target memory, which is nominally 2 KB in our design, that is currently in use by the benchmark is backed by a dynamic allocation on the host. The mapping is very similar to page-based virtual memory allowing the simulator to support a 4 GB address space for the target, but only require the fraction of that 4 GB currently in use to be allocated by RigelSim.

After 150 million target cycles of simulation, which takes roughly four hours to execute on an x86 (host) workstation, the simulator aborted due to the GJK benchmark attempting to read a region of target memory known to contain code. The simulated Rigel system disallows target applications to read or modify code in target memory. The region containing code is located starting at address `0x0000` in the target memory, as shown in Figure 1 in the top left, and is filled before the simulation begins with .text sections from the ELF binary. The ELF

binary is generated by the Rigel compiler from the C code for the benchmark.

Early on in the development of RigelSim, we observed that many bugs in the compiler, assembler, and in benchmark code would manifest as benchmark code or libraries attempting to access regions of memory mapped to code. A trivial, but common, example is a NULL pointer dereference. More generally, bugs of this type would cause stack or heap corruption in the target memory, leading to non-zero values being written over pointers in target memory that are later dereferenced. Sometimes the corruption would lead to aliasing with valid addresses in the target memory, such as having a pointer to one core's stack in memory be replaced with a pointer into another core's stack. The corrupt target memory would eventually cause the benchmark to crash.

The most insidious bugs would cause code to get overwritten in target memory. The modified, corrupt code would later get executed, leading to incorrect results or benchmarks crashing. Supporting self-modifying code was not a design goal, so an assertion was added guarding all accesses to the target memory within the range of code addresses. The guard aborts the simulator when an offending access is discovered. Simple sanity checks based on invariants have proved to be an immense help in finding, isolating, and removing bugs from every piece of the Rigel tool chain. The bug described here was found due to this check.

2.2 RigelSim Memory Model

In this section we describe some of the details of RigelSim's memory model necessary to understand the sections that follow. To simulate the processor efficiently and to ease in debugging, a unified memory model where each address has one canonical value was chosen for RigelSim. Here we describe how RigelSim implements the mapping between target addresses and the host memory.

The memory model tracks the current state of every value at every address in the system. The cache model and interconnect model are responsible for modeling timing. The decoupling between the memory and cache model allows for faster simulation times since data is not moved around between levels of the cache to simulate timing of those actions. Moreover, the decoupling allows for us to separate bugs in the cache and interconnect models from code bugs due to benchmark code corrupting target memory. Once RigelSim has completed modeling the stalling required to get accurate cache and interconnect timing, it issues a request for the address it requires to the memory model. The memory

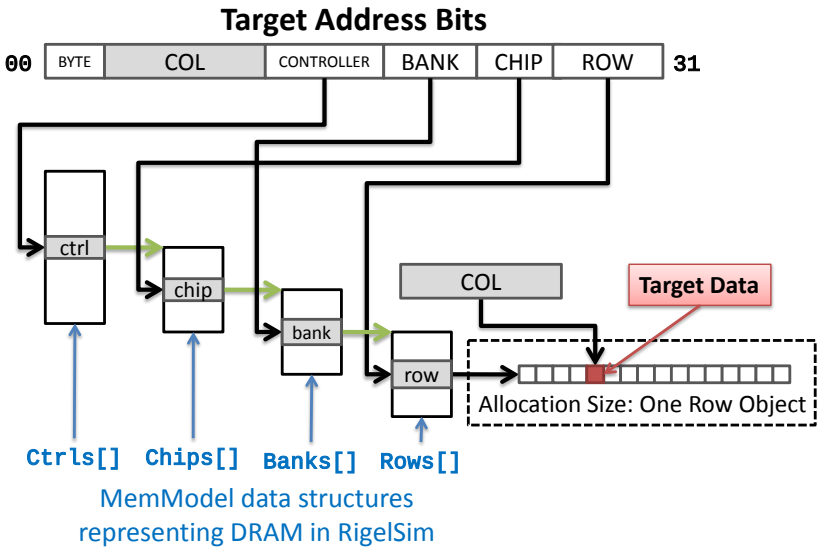

Figure 2: Hashing of the target address into the host address space.

model returns the data, or code on instruction fetches, without stalling the simulation.

The memory model inside of RigelSim dynamically allocates memory on the host side to be used to map regions of DRAM used by the target. Figure 2 shows the multi-level hash of the target address into the regions allocated within RigelSim that hold blocks of simulated DRAM. The allocations are done in 2 KB chunks and may not be contiguous beyond that granularity in host memory. The internal structure of DRAM is used by the simulator as a means to map a target address into a (possibly) incomplete tree structure, with the 2 KB DRAM rows representing the leaves of the tree. In the figure, we show the regions of the target address that are used to index into each array of pointers representing the dimensions of our simulated memory system. The meaning of the various names of each structural element are not critical to understanding what follows, but it is important to note that multiple, chained hashes are employed. When RigelSim starts, before starting the simulation, all `row[]` arrays are filled with NULL pointers. Row objects are allocated only when the index into the `row[]` table finds a NULL pointer present, i.e., the address range covered by that row has not been accessed previously on behalf of the target. The row allocation on the host is oblivious to what target address was other than the bits used for the row index.

For the curious reader, we note that the unified approach causes the simulator's behavior to differ from that of the real chip, but only in the case where software-induced hardware races exist. An example of such a race is when two cores access the same address, with the word located in each of their respective private caches, and at least one is a write. The software has just allowed to two different values for a particular address to exist in the system simultaneously. The race in hardware then becomes which private cache will update the shared cache last and have its value persist. Supporting such races in the simulator is not a goal for RigelSim since well-behaved benchmark code does not use such races and thus their use need not be supported by RigelSim.

2.3 Increased Debugging Support

The simulator produces a debug report consisting of the address of the offending PC in the benchmark, the effective address of the offending access, the line in the RigelSim code where the error occured, what core the access occurred on, and at what simulated cycle when a code word is read or written by a benchmark. The PC tells us where in the benchmark the bad access occurred. By disassembling the benchmark executable, thus translating the ELF binary into Rigel assembly instructions, we can locate what function in the C source code for the benchmark exposes the error, but not necessarily where the fault occurred.

In the process of tracking down this bug, support was added for printing the current stack frame, the register state, and the state of the scoreboard from the core generating the error from within RigelSim. Isolating bugs can often be frustratingly difficult in a large system. Having all of this extra state was necessary for determining at which point the system stopped functioning correctly, even though none of it directly helped to find the bug. When a bug requires a cycle time of three hours from the time a test is started until the results are found, it is important to use any information possible to reduce the number of tests to run. Furthermore, the longer it takes to remove a bug, the more one starts to suspect known-working code. Extra, although ultimately superfluous, debug information was necessary to rule out causes and once added, could be useful in isolating future bugs.

3 The Debugging Process

3.1 Isolating the Bug

At the start of the debugging process, the only information known about the bug was that it was triggered just after initialization code for GJK had finished and it was due to an invalid pointer dereference inside of `atomic_flag_spin_until_set()`, which was taking us three hours to reach from starting a run of RigelSim. Extraneous information known about GJK was that it was the longest running benchmark in terms of simulated cycles, due to a prolonged initialization phase, and that it also had one of the largest code images of any of the benchmarks. The `atomic_flag_spin_until_set()` call is executed on all but one core, which causes them to block until the remaining core, core zero in this case which is doing the initialization work, sets the flag value.

It appeared that the invalid pointer being dereferenced was calculated in the `atomic_flag_spin_until_set()` function, saved on the stack, then repeatedly loaded into a register that was used as the address of the flag to spin on. Each iteration reloaded the pointer value from the stack into a register. It was thought that the initialization routine had a bug and that bug was corrupting target memory causing an invalid pointer to be on the faulting core's stack. As a patch to mask the bug, the `atomic_flag_spin_until_set()` function was rewritten in assembly to eliminate the stack load. The error in the code then migrated to another function.

The one addition to the simulator that did help isolate the bug was a mechanism for printing out a trace of target memory accesses for a range of addresses provided on the command line to RigelSim. The bug that first manifested itself in `atomic_flag_spin_until_set()` moved to the `TaskEnqueue()` function when the former was recoded in assembly. The relevant code inside `TaskEnqueue()` is supposed to access a global data structure that was at a fixed location in memory, i.e., the address is known at compile time. It was clear where the address was generated from looking at the disassembled output of the benchmark binary. It was also clear what the benchmark was supposed to be accessing from looking at the few instructions before the invalid access.

Using the tracing facility, we saw that the `TaskEnqueue()` function was being called numerous times prior to failing. Each of those times it was storing the fixed address on its stack and then loading it back in before dereferencing the pointer. In the last iteration, just before accessing the invalid pointer, the

address was stored to the stack with the correct value and the value loaded back was corrupt; something was corrupting the target memory between the store and the load. Note that even though only a few simulated (target) cycles occur between the store and the erroneous load, potentially billions of host cycles complete, thus opening up a large window in which target memory could become corrupt from within RigelSim.

The trace facility allowed us to see exactly which values were being written to the memory model for the specified range of target addresses. The discord between the load and the most recent store to a given location made it clear that something outside of the memory model was writing over that value inside the simulator. The traces showed that other values were similarly effected in address ranges adjacent to the one holding the corrupted pointer. We assumed a heap allocation bug was the most likely cause.

We had two working theories for what could be causing the problems inside GJK running on RigelSim. The first was that a pointer to one of the regions allocated to DRAM inside of RigelSim aliasing with a previous host allocation that was freed, but still referenced elsewhere. The other theory was that integer overflow was causing a host pointer inside RigelSim to become incorrectly aliased with the allocation for the target address where we were seeing corrupt values being read in from the memory model. Integer overflow occurring only in GJK was a reasonable hypothesis given that GJK was the longest running benchmark, thus increasing the likelihood of overflow in the performance counters that track events on a cycle-by-cycle basis.

We forced the simulator to preallocate all memory that would be used to represent the target's address space as a sanity test. Doing so would force all allocations to be contiguous and persist from the start of execution until completion, thus reducing the likelihood of a use-after-free bug. It was believed that these measures would either mask the bug or cause the triggering action of the bug to change, thus providing added intuition to help further debugging. Even with all of the target memory allocated before the simulator began running the benchmark, GJK still failed at the same location within RigelSim at the same PC value after running for 150 million simulated (target) cycles. The Electric Fence library was also used in an attempt to uncover heap allocation bugs. The library links with an application, RigelSim in this case, and hooks all dynamic memory allocation and deallocation calls to check for leaks and invalid references. Due to the increased number of pages required by Electric Fence, the heap allocator failed after about three hours of running, executing

only 77 million simulated cycles, without finding any memory leaks or invalid heap accesses in RigelSim.

We replaced all occurrences of 32-bit types in the RigelSim statistics collection code with 64-bit types to make overflow effectively impossible, thus allowing us to test our hypothesis regarding integer overflow. The overhead in terms of memory and possible execution efficiency losses on 32-bit platforms was a concern, but the overheads were found to be negligible . Even with the change, the bug remained unaffected. We tried running on both 32-bit and 64-bit host systems which also had no affect. Disabling compiler optimizations when building RigelSim had no affect. Perturbing external factors such as runtime environment and compiler configuration would not have fixed the bug, but the hope was that additional insight could be gained by observing the perturbations' effects on the bug. However, due to invariance under platform changes and even code changes, a logical error in the simulator became the new working theory. The location of the logical error, however, was still unknown.

3.2 Debugging With GDB

We were unable to reason about what was causing the bug, but it was deterministic and affected one particular target address. We also assumed it only affected a single host address as well. We had not used a debugger up until this point because the bug was causing the simulated benchmark to crash *not the simulator itself*. Furthermore, the overhead associated with running the code inside the debugger only added to the turnaround time on trying new debugging approaches.

We attempted to use the watchpoint facility of GDB to observe the host memory backing the target memory that was being corrupted. A watchpoint allows for the host processor to track accesses to a specific range of addresses in hardware, with near zero overhead, and trap when the host processor issues a load or store to that range on the behalf of our application. Placing a watchpoint on the location holding the target memory value would allow us to not only see the load and store transactions between the core model and the memory model, which we were already able to see from the memory traces, but also would allow us to find any other region of the code that was erroneously updating that value.

Placing a watchpoint on the address proved to be harder than expected. The actual allocation of the target's memory is buried beneath multiple levels of calls through the tree structure representing DRAM inside the memory model

as shown in Figure 2. The allocation point on the host side has no notion of what target address it is mapped to due to the hashing of the target address into the tree representing DRAM that occurs. We attempted to get access to the host address backing a target address, which we would need in order to add a watchpoint, by using static member functions of the memory model. The static member function would do the address-to-allocation mapping from within GDB taking a user-provided target address on the GDB command line and translating it into the address within RigelSim that it maps to. Every attempt to set a hardware watchpoint to the allocation from within GDB returned an error.

We could only insert a software watchpoint, which was an untenable solution. Software watchpoints have orders of magnitude higher overhead compared to hardware watchpoints since they single step through the execution of the application being debugged, which is RigelSim in this case, checking each *host* cycle for memory updates. We ran with a software watchpoint on for over ten minutes without seeing the simulator advance a single target cycle; natively the simulator runs at thousands of target cycles per second.

We had RigelSim dump the host address of the allocation that held the region of the target's memory that was getting corrupt. We were only interested in a target single address which, due to preallocation of the target memory, was always in the same place in host memory. We used the host address inside GDB to manually force a watchpoint to be set. We isolated the last valid access to the allocation and set the watchpoint after a bit of manipulation and setup. The watchpoint triggered at the same location within RigelSim, but to a different target address. That is when the bug became clear: Our assumption regarding the one-to-one mapping of target to host addresses was flawed—multiple target addresses were mapping to the same host address, causing aliasing to occur. Figure 3 illustrates how two addresses would alias causing writes to one variable inside the target to overwrite the value of another variable incorrectly.

3.3 Removing the Bug

Recent changes to the simulator were performed to correct the mapping of target addresses to global cache banks. Those changes inadvertently altered the mapping of target addresses to memory and led to the top bit of the target address space being ignored. Two addresses, in this case one for the stack and another that pointed to part of the library code used for work allocation on Rigel, now aliased and would overwrite each other. We had RigelSim attempt

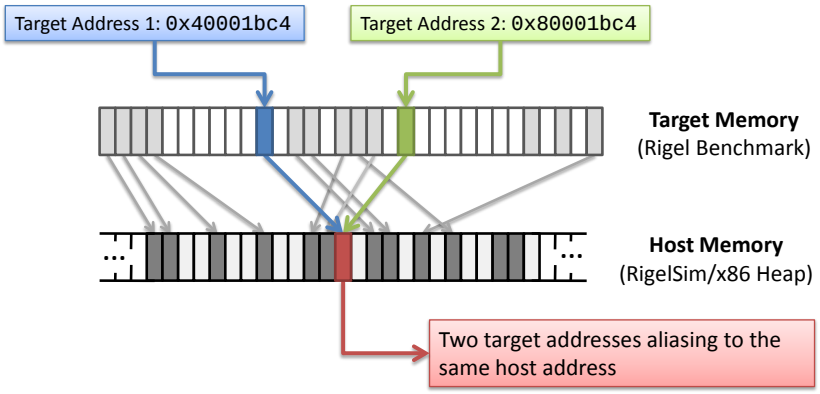

Figure 3: Two target addresses (incorrectly) aliasing to the same host allocation

to access every block of memory starting at address zero, checking for the proper initial state of all zeros and then poisoning the block with some other value as a simple test. The first location where the poison value is read would cause an error to be printed out with the address, thus isolating the point where wraparound begins to occur. After accessing a gigabyte of memory, which took less than a minute to reach, the simulator found aliasing.

Fixing the bug involved rewriting the hash functions that map target addresses into the memory model's structure that holds allocated target memory. The groupings of bits shown in Figure 2 need to be set so that each bit is only used once in the mapping process. The CONTROLLER and CHIP numbers are runtime constants, which means the hashing routines are a function of command line parameters passed to RigelSim, making the mapping non-static. Furthermore, the mapping routines are used in the interconnect and cache model to determine how to map requests for memory onto the appropriate channels and to the caches that sit in front of the DRAM controllers, making the functions vulnerable to changes in other parts of the code. The fix ultimately took about two hours and is now easily regression tested with the simple address space enumeration/poisoning technique described earlier.

4 Lessons

What follows are some of the lessons we learned from the bug documented in this case study and from our debugging experience with RigelSim in general.

An overreaching lesson is that the methods and techniques for locating and removing any particular bug can help locate future bugs if incorporated into the development and testing methodology of the project. It is hard to provide a general algorithm for debugging, but through experience and case studies, it is possible to collect a set of patterns and tools that can be applied to other software bugs and to reduce the time to isolate and remove those bugs.

4.1 Isolation vs. Location vs. Masking

The end goal of a debugging session is to *locate* a bug that is causing an error in a system and remove it from the component in which it exists. For large multi-module systems, such as the Rigel tool chain used in this case study, locating a bug is less of an issue than narrowing down, or *isolating*, which piece of the system is responsible for generating the bug. Isolation becomes even harder when the systems are designed and implemented by multiple individuals where incomplete information, poor communication, lack of documentation, and unwillingness to accept responsibility can cause issue.

Many bugs in the Rigel tool chain do not become activated until the code is run, either in a target benchmark or in the host system. Even then, the bugs only manifest when certain conditions are met, such as the number of cycles simulated or a particular input set is selected. A bug in the Rigel compiler could cause a library to have a latent fault that is then only triggered when that library is linked with a new benchmark, and only then when it is run on the simulator for 100 million target cycles, which can take many hours as was the case with the bug in this case study.

The process of isolation is critical because it is foreseeable that the team responsible for the compiler (and the bug!) is not the same as the team responsible for writing code or maintaining the simulator (who found the bug). When bugs cross modules, time can be wasted locating a bug in a module where a bug does not exist. Time can also be wasted finger pointing as other groups do not want to take ownership of bug reports. The faster a bug can be isolated, the sooner people with the skills available to tackle the problem can be deployed to the module where the problem actually exists, not just where it is observed.

One other point to note is that fault *masking* can be effective in both understanding the nature of a bug, so that it can be later removed, and for serving as a stopgap measure when the buggy software needs to be available sooner than the time horizon for removing the bug. The bug described in this case

study was due to memory corruption leading to an invalid pointer dereference. An attempt was made to remove the stack access that generated the corrupt address. The goal was to get the benchmark working again, not necessarily to remove the bug.

The bug could in fact have been in the compiler and manipulating the benchmark assembly code by hand to remove the bug inserted by the compiler would have temporarily solved the problem. The root of the problem, i.e., the compiler, would not be fixed, but the quick fix would have allowed us to continue running simulations. Even though the bug masked at `atomic_spin_until_set` simply migrated to another function, the effect of masking allowed us to better understand what the bug was sensitive to, in this case memory accesses, which inevitably led to us locating the bug.

4.2 Self-checking

Systems should be built to be self-checking when possible. A common form of self-checking are assertions that force the system to abort should a constraint known by the programmer be violated by the system at runtime. If the self-checking overhead is low, it can be done on-line and continuously without any programmer or user intervention. On-line checking provides high coverage and can isolate bug sooner than would otherwise be possible. If we had not run GJK with the parameters we selected, we would have not found the bug described in this case study, possibly only to see it surface at a later time when we did not have the ability to correct it. With on-line checking in place, the possibility of bugs covered by those checks being manifested at an inopportune time, or worse, not detected at all while allowing for erroneous results to be generated by the system, is greatly reduced.

Another key issue is that self-checking should be simple and robust to changes in the code. If the checkers are not simple, there is a good chance that the checkers might be incorrectly implemented itself leading to missed bugs. Another concern when adding on-line checkers is the chance of false positives, which can lead to user frustration. The frustration results in programmers and users turning off checking and thus bugs being masked in development only to manifest in production or bugs becoming difficult to isolate as early warning provided by the self-checkers will not occur.

The ability to access all of memory without aliasing in RigelSim has a simple, albeit slow, check for correctness that enumerates all locations in memory. The

benefits of the approach include its simplicity and the ability to test the module directly using the same code path as a production run would use. Other methods that used external hooks to access the low-level memory could be used, but would cause the test runs and production runs to use separate code paths, potentially masking bugs. The check was made into a regression test that is run periodically, but would be too costly to run every time RigelSim is executed.

One method could be to prove that our hash functions can map the whole address space. The proof would need to show that the set of hash functions that map the target address space into the simulated DRAM are both one-to-one and onto. The benefits of such an approach are that it can be low-cost in runtime overhead since it is a static check of the hash functions in the code. The deterrent to using this method is that the proof may be difficult to generate. Even with a proof in hand, checking for equivalence between the model used in the proof and the simulator code is non-trivial. If the issue of equivalence checking were overcome, the proof may only hold for one set of mappings and thus would not be robust to changes in the RigelSim code, which could change frequently.

4.3 Regression Testing

The simple memory checker for enumerating all of memory in search of aliasing is too costly for production runs. The high cost is due to the large amount of memory that must be used to track all of the blocks of memory already seen by the checker. However, the few minutes it adds in overhead make it ideal to do as a regression test or prior to checking in changes to the hash function that maps the address space to the simulated DRAM. Once the effort has been sunk into isolating and removing a bug, it should be effort leveraged for stopping the bug from reappearing. Furthermore, a check for one specific bug can also provide greater coverage for unforeseen bugs in the system such as the code address range guards placed in RigelSim, which lead to the isolation of the bug described here.

4.4 Robustness and Determinism

Systems should be built to either be robust to failures or to fail quickly and deterministically. The bug described here was relatively deterministic: we could run RigelSim with different configurations, with code changes due to compiler optimizations being turned off, and with debugging hooks such as GDB and

Electric Fence enabled and still reach the bug. If this had not been true, the debugging experience we describe may have been made much more difficult. However, deterministic bugs are not inherently good, but rather tell the engineer attempting to remove the bug traits about the bug. In our case, the deterministic nature of the bug was due to a logical error in the design as opposed to a misuse of an API or memory corruption due to overflow or dynamic memory mismanagement, which would have likely caused the bug to migrate during the processes of isolating it.