

I/O Branch Bugs on Intermittent Systems

Milijana Surbatovich

Emily Ruppel

1 Introduction

2 Background

...

2.1 A discussion of non-idempotent behavior in intermittent programs

In Figure 1, we present a flow chart of the taxonomy of non-idempotent behaviors and their potentially buggy consequences.

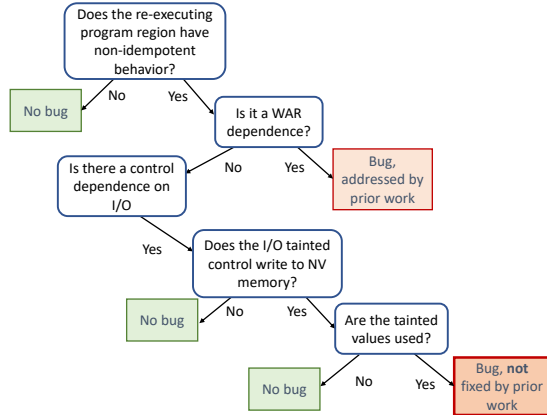


Figure 1: Taxonomy of non-idempotent behavior and consequences

Question 1: Does the program have non-idempotent behavior? Idempotent code can be re-executed arbitrarily without any changes to the visible output. [Find a citation??] Most new bugs introduced by computing under intermittent power are due to memory inconsistencies caused by the unexpected re-execution of code regions. If a program does not exhibit non-idempotent behavior, it will not have any of the bugs we are studying.

Questions 2 and 3: What is the cause of the non-idempotency? Non-idempotent behavior generally has two sources: write-after-read (anti) dependencies and I/O

operations.¹ Code re-execution after a WAR dependence creates direct data-flow that did not exist in the original program, and I/O creates effects that cannot be undone. Bugs off of WAR dependencies have been extensively studied in prior work, and several runtime systems exist to fix them [3] [2] [1]. While there could be other bugs stemming from the use of irrevocable I/O operations, this paper specifically explores memory inconsistencies stemming from branches that are control dependent on non-idempotent input.

Questions 4 and 5: When can an I/O dependent branch cause a bug? Not all I/O dependent branches necessarily lead to buggy behavior. To make memory be in an inconsistent state, non-volatile variables must be written to on at least one side of the branch, as they will still be visible to the program on reboot. Furthermore, on a power fail and re-execution of the non-idempotent region, such a tainted variable has to be the reaching definition for some use, either in expected program execution or along the back-edge introduced by re-executing. We discuss this further in Section [false positives]. If the targets of the I/O dependent branch do write to non-volatile memory, however, and there is some use of these tainted variables, this can cause bugs that have not been addressed by prior work.

3 System Overview

The overarching algorithm is fixed-point, iterating through the functions, summarizing new information about the io flow, and iterating through functions again, until there is no new information to be gleaned.

To start with, we don't know where the io enters the program, so all the functions are put in the stack, with no io parameter—the io call/instruction is found internally, based on annotated information in the source code. Once the IO source is found, we call a function that traverses the local data and control flow chain of the io source. In the chain, there are some instructions of interest—namely,

¹Besides WAR operation and I/O, there are some less common causes of non-idempotency, such as pointer-based stack operations, as noted in prior work by Van Der Woude et al. [3].

branches, which can lead to the bug of mismatched nv write sets, and instructions that result in io tainted values flowing out of the current function that we're examining.

Traversing the data flow primarily uses LLVM's built-in def use chains. From the definition of the tainted variable, we follow the uses in a depth first search, adding on new definitions to the stack. From the traversal function call, we return the list of interesting instructions found. If the instruction was a branch, we examine the non-volatile (i.e.,globals) write sets to see if there is the possibility for the bug. Additionally, the non-volatile variables (used locally and in nested function calls) and local volatile variables written to are control-dependent on the io tainted variable, so we add the globals to the summary information and continue the local traversal with the newly tainted control dependent variables.

For the other types of instructions returned from the traversal, we use the information to add to our list of functions to still be examined to reach fixed point.

Values can leave the local function by:

- being returned from the function
- being passed into a global variable
- being passed as a parameter into another function
- being stored into a call-by-reference parameter (which may be global itself)

If the tainted value was returned from the function F, we add all the callers of F to the stack, with the value where the return was stored as the tainted io value. (When adding new functions to the examination stack based on summary information, we pass in the tainted value as a parameter instead of searching for the annotations, but then search the chain in the same way.)

Likewise if the tainted value was passed into a global, we find the uses of the global, the parent Function of those uses, and add those to the stack, with the global as the tainted value. If the io was used as a function parameter, we find the callee function and the local version of the argument.

If io was stored into a pass-by-reference parameter, we find all the caller functions and their local value that the argument aliases to. (If the reference para refers to a global, we also add that global to the summary information.)

We add all these new function/tainted value pairs to the stack, and continue to iterate. (We do keep track of what pairs we have already examined, to prevent infinite looping). Eventually, there will be no new information being returned from the traversal function, and we have examined all the functions to which the tainted io has spread.

3.1 NV Write-set algorithm

To help with handling arbitrary complexity of control flow within the branches off of io, we precompute the may write sets of each function before hand, using a bottom up examination of the call graph.

What stores we decide to add the write sets does effect what possible bugs will be reported. For instance, one complexity that must be considered is how to handle stores on branches. Say we have a simple program with two nonvolatile variables, nv0 and nv1 that branches on an io tainted variable. On one side of the io branch we write to nv0 and nv1. On the other side, we write to nv0, but then we branch again. On one side of the branch we write to nv1, but on the other side we don't. Thus, in this example, one execution could have the io branch writing to the same nv set on both sides, and another execution doesn't.

There are a few ways to deal with these branches: don't add any stores within the branch basic blocks to the write set, add all the stores to the write set (what we do right now), or make two versions of the write set, adding a different side of the branch to each one. This last option is perhaps most attractive for accuracy, but it is difficult to handle arbitrary complexity, and has a much higher storage overhead.

Another problem with the first two approaches is that if we adjust the above example a little, we can have an io branch where both sides write to nv0, and on the second branch, one side also writes to nv1. In this case, adding all the stores with the branches detects more bug possibilities, whereas ignoring the branch under reports. So neither of the simpler solutions can be said to definitively under or over report bugs.

3.2 Filtering Pass

//TODO

3.3 High-level, English explanation of what the tool detects

- The final bug is always the disparate nv write sets after a branch off of an io tainted value
- We track the io through explicit (data) and implicit (control) flow
- Tainted io can leave a function through function calls, return values, pass by ref, and globals
- Initial io into the prog has to be annotated (not good or bad, just a thing)

3.4 Possible limitations of the tool

- Precomputed write sets
- Overhead?
- How complex can the src code get? – we should probably impose a limitation on branching levels or function depth
- Are there any other suspicious bug patterns? i.e., completeness
- Over or under report bugs? Right now tending towards conservatism

4 Evaluation

5 Implications

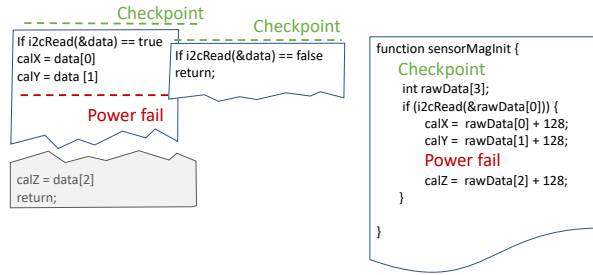


Figure 2: Bug in magnetometer initialization. Power failing while updating the calibration fields can cause the calibration data to become inconsistent, corrupting any future magnetometer reads that use the calibration.

6 Surprises and Lessons Learned

7 Conclusion

References

- [1] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 575–585, New York, NY, USA, 2015. ACM.

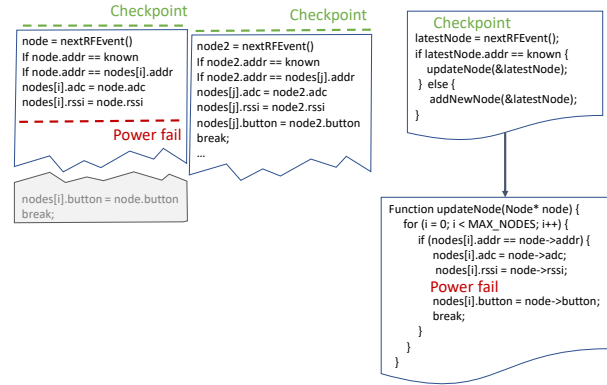


Figure 3: Bug in WSN concentrator. Power failing while updating the node structure can cause the node list to become inconsistent, corrupting the payload ("button" field) or using out-of-date timing information

- [2] K. Maeng, A. Colin, and B. Lucia. Alpaca: Intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 1(OOPSLA):96:1–96:30, Oct. 2017.
- [3] J. Van Der Woude and M. Hicks. Intermittent computation without hardware support or programmer intervention. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, page 17, 2016.

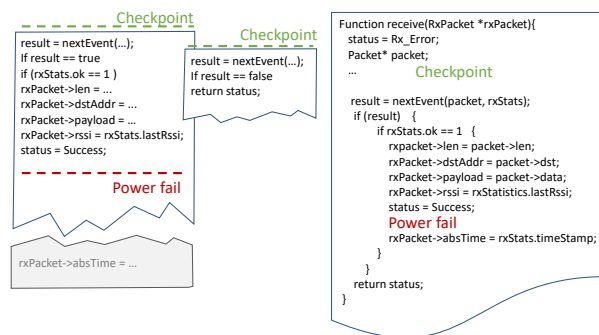


Figure 4: Bug in RF EasyLink Receiver. Power failing before returning a correctly read packet can cause the status field to be set to success, even if the next event fails. The function will return an incorrect status value, potentially crashing the larger application.