CSE 231—Advanced Operating Systems "Failure Oblivious Computing"

Andrew Quinn

Summary. This paper introduces the idea of failure oblivious computing (FOC) [3] Essentially, the paper investigates the applicability of ignoring memory errors by performing "reasonable" default behavior when an invalid read or write occurs. Their solution works well across a number of real-world web-servers, largely because these applications have short "error propagation", i.e., errors effect a relatively small portion of an execution immediately after the error, but do not effect the long-term behavior of the application.

Perhaps the most confusing part of this paper is the actual system design and implementation. They used all of about a page to describe this section; probably because FOC mostly builds on preexisting work into safe-C. I won't describe the actual implementation in detail, except to say that it builds on idea that "all pointer calculations should point to the same object as the pointers that were used to derive them (e.g., `a * ptr + b` should point to `ptr`) [1, 4]. These approaches then track the object that each pointer calculation points-to, and identify errors when the calculation does not lie in the bounds its object.

Discussion. This paper is a highly controversial idea that was reportedly very contentious at the OSDI '04 conference. My position is that it is a totally reasonable approach (if, admittedly, a dangerous one) because currently, the C/C++ spec defines the memory errors that FOC targets as undefined behavior. That is, we currently have *no* guarantees about what happens when a program performs a buffer overflow [1]. Undefined behavior can be a

[1] I'm not certain of the details of memory errors, but other undefined behavior, data races, are so unsupported that the compiler can actually give you back an arbitrary program if your program has a data race!

huge problem [5] for modern systems, and some guarantees are clearly better than no guarantees, right?

This tough to answer, especially for current applications. For example, Linux provides a mechanism to catch and handle segmentation faults (i.e., signal handlers), which might obviate the need for FOC. Moreover, FOC also has a major potential impact on debuggability, especially in distributed contexts. Supposing that there are multiple failures that were handled by FOC, how do I know whether a downstream failure occurred only because of FOC's handling of an earlier failure? This is particularly problematic in the context of a large distributed system, where tracing the causality of application behavior can be next to impossible. Logging all failures that FOC handles seems clearly insufficient.

In class, we outlined an alternative approach based upon speculation and checkpointing. Essentially, our idea was to checkpoint the application at every user request. If request processing fails, then you can simply rollback to the earlier checkpoint. Unfortunately, it is not obvious when a request begins, so the approach will likely instead need to checkpointing and potentially rollback after each input to the server. There have been systems with similar approaches to this, such as EVE, which used speculation for state machine replication of multi-threaded programs [2]. However, general adoption seems difficult since it is unclear whether a "good" rollback point can be identified after a failure in general.

There are other "seat-belts" that provide similar failure obliviousness (but for other classes of failure). Of particular interest is Frost [**?**], which builds on the DoublePlay work by simultaneously executing two versions of a program with each version following a *complimentary schedule*. The key insight is that deviations between two complimentary scheduled executions only arise because of harmful races, and you can usually determine which version is "correct" by looking for obvious failures (e.g., segfaults).

References

[1] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *AADEBUG*, 1997.

[2] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for

multi-core servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 237–250, USA, 2012. USENIX Association.

[3] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*, OSDI'04, page 21, USA, 2004. USENIX Association.

[4] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[5] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 260–275, New York, NY, USA, 2013. Association for Computing Machinery.