CSE 231—Advanced Operating Systems "Eraser"

Andrew Quinn

Software Reliability. Generally, in software reliability ask the question: "How do we make our systems *behave correctly*?". For this class, we'll focus software bugs as opposed to fault tolerance work, which usually focuses on hardware failures.

A bit of terminology that we will use:

- **Failure** When a program's external output differs from its specification.
- **Fault** The algorithmic cause of a failure. We will use the term **bug** interchangeably with fault.

The community studies and improves reliability in a number of ways, including:

1. **Bug-Finding in large systems**. Many systems works study how to find bugs in large systems. Usually, these papers also survey and seek to understand the properties of the bugs they find. Example work includes: Bugs as Deviant Behavior [1], Crash-consistency detection [5], data race detection [2], and today's paper, Eraser [7].
2. **Guardrails**. Much like guardrails prevent vehicles from falling off of a road, some systems have been designed to prevent or reduce the impact of bugs in production. Such systems include Nooks [9], Failure Oblivious Computing [6], and Frost [10].
3. **Better Engineering**. Many works aim to provide fundamentally better engineering practices to reduce the incidence of bugs in software systems. Most of the research in the community in this area is in formal verification, such as IronFleet [3], the push-button verification work [8], and the SeL4 [4].

Summary. Eraser introduces the lockset algorithm to find concurrency issues in software. Their primary argument is that most software uses a locking mechanism to mediate access to shared memory, so many bugs can be found by simply tracking the properties of locking within an application.

I love this paper for our class because it is a great introduction to so many aspects of bug finding, including soundness vs. completeness, dynamic vs. static, and fault-detection vs. failure-detection.

We'll go through each of these ideas in the next few paragraphs. But, first let's give ourselves a model for thinking about a program and its execution. A program, `P`, is a blob of text written in a particular grammar (for our purposes, we won't consider programs that do not compile). An execution of the program, `E`, is the result of passing a specific input to the program. We can think of the execution as a trace of events. N.b., there are weak memory consistency models that cannot be totally ordered and thus cannot be modeled in this way, but these details are not important for our discussion. **??** provides pictorial representations of these concepts.

soundness vs. completeness. There are two goals that a bug detector should try to optimize. First, a bug detection should be **sound**, i.e., ideally, it should report no false positives, or bugs that are reported by the tool but are accepted behavior. Additionally, a bug detection tool should be **complete**, i.e., ideally it should incur no false negatives, or bugs that are in the program, but not reported by the tool. Unfortunately, bug detection is almost always impossible since it is involves inspecting an arbitrary program and is thus equivalent to solving the halting problem.

Thus, most systems trade-off soundness and completeness, and many trade-off both. For example, at numerous points in the paper, Eraser uses a heuristic that incur false negatives in the interest of reducing false positives. There is no correct answer in determining whether to favor soundness and completeness; no matter what decision your tool chooses, reviewers will likely leave upset.

dynamic vs. static detection. There are essentially two broad classes of bug detection: dynamic tools, which trace an actual execution to find bugs, and static tools, which inspect the source code of a program to find bugs. In general, static analysis tools can be more complete, since they can inspect all possible actions that a program might take. However, static program analyses

struggle due to the combinatorial number of potential program behaviors, so static analyses are almost always unsound and report false positives (n.b., the alternative is a static analysis with no false positives, but many false negatives since it has limited understanding of the target program). Dynamic program analyses have runtime information and thus make better decisions, but inherently are incomplete since they only view a single program execution at a time.

fault-detection vs. failure-detection. Dynamic detection tools can use a few different approaches for bug detection. The first is what I call failure-detection: the testing tool can inspect program output to find a failure. Such a tool can look for obvious failures (e.g., segmentation faults or failed assertions) or can look for specification specific issues (e.g., many file system tools understand the intended semantics of POSIX file system operations). Models of the intended specification for a program can be difficult to build for an arbitrary application, so many developers instead inspect the trace of actions taken by the program to detect a bug (essentially, detecting the fault in the program). The challenge is that not all "faulty" behavior will inherently lead to a failure in all programs. For example, while a invalid pointer derference is undefined behavior in the C/C++ specification, a program could include a signal handler that allows enables correct output in the case of a segfault and survive such canonically faulty actions.

References

[1] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 57–72, New York, NY, USA, 2001. Association for Computing Machinery.

[2] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *OSDI*, volume 10, pages 1–16, 2010.

[3] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th*

Symposium on Operating Systems Principles, SOSP '15, page 1–17, New York, NY, USA, 2015. Association for Computing Machinery.

[4] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[5] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 433–448, Broomfield, CO, October 2014. USENIX Association.

[6] 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.

[7] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, nov 1997.

[8] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 1–16, Savannah, GA, November 2016. USENIX Association.

[9] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03,

page 207–222, New York, NY, USA, 2003. Association for Computing Machinery.

[10] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 369–384, New York, NY, USA, 2011. Association for Computing Machinery.