REPT: Reverse Debugging of Failures in Deployed Software

Talk by

Dixith Reddy Nayeni

Introduction

- Debugging software failures in deployed systems is important because they impact real users and customers.
- However, debugging such failures is notoriously hard in practice because developers have to rely on limited information such as memory dumps.
- The execution history is usually unavailable because high-fidelity program tracing is not affordable in deployed systems.
- The paper presents REPT, a practical system that enables reverse debugging of software failures in deployed systems.

Overview

- REPT (Reverse Execution and Prediction Tool), is a system for debugging software failures in deployed systems.
- It is a new approach to do reverse debugging in deployed software by combining online lightweight hardware tracing of a program's control flow with offline binary analysis to recover the program's data flow.
- It addresses three technical challenges in binary program analysis, including handling irreversible instructions, memory writes to unknown addresses, and shared memory accesses, by combining backward and forward analysis, monitoring memory accesses, and including context-sensitive analysis.
- The paper describes the implementation and deployment of REPT, which leverages Intel Processor Trace to log control flow and timing information and performs offline binary analysis using a memory dump and trace input.
- Paper evaluates REPT's accuracy, efficiency, and effectiveness for debugging failures, with results showing high accuracy in recovering data values compared to Time Travel Debugging.

Existing Approaches

- Existing approaches can be classified into two categories.

1. **Automatic root cause diagnosis**
   - This approach aims to automatically determine the statements responsible for a program failure.
   - However, these systems are not practical for use due to various limitations such as code modification requirements, inefficiency in handling complex software, or being limited to a subset of failures.
2. **Failure reproduction for debugging**
   - This approach enables developers to examine program inputs and states that lead to failures.
   - However, existing techniques such as symbolic execution, model checking, state-space exploration, and record/replay systems have limitations such as requiring heavyweight runtime monitoring or incurring prohibitive overhead.

- Similar/related applications are error reporting services deployed by major software vendors and open-source systems to collect data about failures in deployed software and analyze them.

Similar applications & Current Limitations

- Similar/related applications are error reporting services deployed by major software vendors and open-source systems to collect data about failures in deployed software and analyze them.
- The most advanced bug diagnosis system deployed in production, RETracer, is limited to triaging failures caused by access violations.

- Current Limitations

1. High runtime performance overhead.
2. Inaccurate and inefficient recovery of the execution history.
3. Need for code modification.
4. Limited applicability to only certain classes of bugs.

Uniqueness of REPT

- REPT is better than the existing ones because it leverages **hardware tracing** to record a program's control flow with low-performance overhead and uses a **novel binary analysis** technique to recover data flow information based on the logged control flow information and the data values saved in a memory dump.

- This combination of control flow and data flow information enables effective reverse debugging.

- It uses a new binary analysis approach that combines forward and backward execution to iteratively emulate instructions and recover data values.

- REPT is implemented in two components that have been deployed on hundreds of millions of machines as part of Microsoft Windows, making it a practical solution for reverse debugging in deployed systems.

REPT explained

- The goal of REPT is to enable reverse debugging with low runtime overhead by using hardware support to log control flow and timing information and using a new offline binary analysis technique to recover data values in the execution history.

- The paper makes three design choices: relying on a memory dump, performing the analysis at the binary level, and using concrete execution instead of symbolic execution.

- Deployment of REPT can be integrated into existing software development workflows.

- The tool is deployed by running it on a memory dump of a failed process, which is collected and analyzed offline.

- The output of REPT is the recovered execution history of each thread, which can be used to identify the root cause of a failure.

Challenges of REPT

- The first challenge is handling irreversible instructions.
- The second challenge is handling memory writes to unknown addresses, which REPT solves by using error correction.
- The third challenge is correctly identifying the order of shared memory accesses in the presence of concurrent memory writes from multiple threads.

Strengths

- The results show that REPT achieves high accuracy, with most cases having a percentage of correct register usage above 90%.
- Performance overhead is below 2%.
- The deployment of REPT proves that its performance overhead is acceptable in practice, particularly when it is selectively turned on for a program on a user machine.
- Out of 16 bugs that were considered, REPT is effective for 14 bugs.
- When deployed in Microsoft, a two-year-old bug was fixed in just a few minutes with REPT.

Weakness

- First, the control flow trace may not be long enough to capture the defect (e.g., the free call is not in the trace for a use-after-free bug).
- Second, data values that are necessary for debugging the failure are not recovered (e.g., the heap address passed to the free call is not recovered for a use-after-free bug).
- Large circular trace buffer cannot be simply used to solve this problem as the data recovery accuracy decreases when the trace size increases.
- The current implementation of REPT only supports reverse debugging of user-mode executions.
- Kernel-specific artifacts need to be properly handled such as interrupts to support reverse debugging of kernel-mode executions.

Future Work

- There is a need to log more data than just the memory dump.
- It is an open research question to identify a good trade-off between online data logging, runtime overhead, and offline data recovery.
- A potential direction is to leverage the new PTWRITE instruction to log data that is important for REPT's data recovery.
- The evaluation of REPT has been focused on software running on a single machine. When developers debug distributed systems, they usually rely on event logging. It is an interesting research direction to study how program tracing can be combined with event logging to help developers debug bugs in distributed systems.
- REPT was not applied to mobile applications because there is no efficient hardware tracing like Intel PT available on mobile devices.

Conclusion

- REPT is a practical solution for reverse debugging of software failures in deployed systems.
- REPT can accurately and efficiently recover data values based on a control flow trace and a memory dump by performing forward and backward execution iteratively with error correction.
- It has been implemented into the ecosystem of Microsoft Windows for program tracing, failure reporting, and debugging.
- Experiments show that REPT can recover data values with high accuracy in just seconds, and its reverse debugging is effective for diagnosing 14 out of 16 bugs.
- With REPT, the hope is that one day developers will refuse to debug failures without reverse debugging.

References

- REPT: Reverse Debugging of Failures in Deployed Software by Anwar & Sally - https://www.usenix.org/system/files/osdi18-cui.pdf