TUHH
Hamburg
University of
Technology

Bachelor Thesis

Robin Willenbrock

Static Detection of Data Races in Interrupt-Driven Software Using Reduced Inter-Procedural Control Flow Graphs

June 19, 2024

supervised by:
Ulrike Engeln

Hamburg University of Technology (TUHH)
Technische Universität Hamburg
Institute for Software Systems
21073 Hamburg

STS
Software
Technology
Systems

Contents

List of Figures

1 Introduction

2 Background

2.1 Interrupt-Driven System

An interrupt-driven system is a type of computing architecture where the flow of execution is altered by asynchronous events known as interrupts. These interrupts can be generated by hardware devices, software conditions, or external signals, prompting the processor to suspend the current tasks and execute an interrupt handler or interrupt service routine (ISR). Interrupt-driven systems are prevalent in real-time operating systems (RTOS), embedded systems, and applications where timely responses to external events are critical (Wang et al., 2020).

Interrupts can occur at any time, making the system highly responsive but also challenging to manage due to the unpredictability of execution flows. Interrupts are often prioritized, with higher-priority interrupts being able to pre-empt lower-priority ones, ensuring that critical events are handled promptly. Handling an interrupt involves saving the current state of the processor and switching context to the ISR, which introduces complexity in maintaining consistent system state. Multiple ISRs can potentially run concurrently with the main program, leading to concurrent execution scenarios that must be carefully managed to avoid conflicts and ensure data integrity (Wang et al., 2020).

2.2 Reduced Inter-Procedural Control Flow Graphs

A Control Flow Graph (CFG) is a representation of all paths that might be traversed through a program during its execution. In the context of inter-procedural analysis, an Inter-Procedural Control Flow Graph (ICFG) extends this concept by incorporating the control flow between different procedures or functions in a program. Reduced Inter-Procedural Control Flow Graphs (RIPCFG) are optimized versions of ICFGs, designed to simplify the analysis while preserving essential information (Engler and Ashcraft, 2003).

RIPCFGs utilize various techniques such as node merging, edge contraction, and elimination of non-essential nodes to reduce the complexity of the graph without losing critical control flow information. By reducing the size and complexity of the control flow graph, RIPCFGs make it feasible to analyze large and complex software systems, which would be computationally prohibitive with full ICFGs. RIPCFGs retain enough detail to facilitate accurate static analysis, including data flow analysis, control flow analysis, and the detection of potential data races and other concurrency issues. The reduced size and complexity result in more efficient analysis algorithms, enabling faster detection and resolution of issues in the software (Wang et al., 2020).

2.3 Data Races

Data races occur when two or more threads access shared data concurrently, and at least one of the accesses is a write. Data races can lead to unpredictable and erroneous behavior, making their detection and resolution a critical aspect of concurrent programming (Flanagan and Freund, 2009).

Data races arise in systems where multiple threads execute without proper synchronization, leading to conflicting operations on shared data. The outcome of a program with data races is non-deterministic, as the order of execution and interleaving of operations can vary, leading to inconsistent results and hard-to-reproduce bugs. There are different types of data races, including write-write races, which occur when multiple threads write to the same variable simultaneously, and read-write races, which occur when one thread reads a variable while another thread writes to it concurrently. Various static and dynamic analysis techniques are used to detect data races. Static analysis involves examining the code without executing it, while dynamic analysis monitors the program during execution to identify potential race conditions. Proper use of synchronization mechanisms, avoiding shared mutable state, and designing thread-safe data structures are key strategies to prevent data races (Engler and Ashcraft, 2003; Flanagan and Freund, 2009).

2.4 Static Detection of Data Races in Interrupt-Driven Systems

In interrupt-driven systems, the asynchronous nature of interrupts and the concurrent execution of ISRs and the main program introduce significant challenges in ensuring data consistency and detecting data races. Static analysis techniques, particularly those using RIPCFGs, offer a promising approach to identifying potential data races without the need for exhaustive testing or runtime monitoring (Wang et al., 2020).

The approach involves constructing RIPCFGs for the program, including both the main code and ISRs, capturing the control flow and potential interactions between them. By analyzing the RIPCFGs, paths where shared data is accessed concurrently without proper synchronization can be identified, indicating potential data races. Integrating the static analysis tool with the development workflow enables continuous detection and resolution of data races during the software development lifecycle, improving the reliability and correctness of interrupt-driven systems (Wang et al., 2020).

3 Implementation

3,5 Wochen

4 Evaluation

1,5 Woche

5 Conclusion

Indroduction+Conclusion und Allgemeine Überarbeitung 0,5 Woche 1 Wochen Korrekturlesen und Einarbeitung =9 Wochen bei Vollarbeitszeit an BE

Bibliography

Lightweight Data Race Detection for Production by Swarnendu Biswas, Man Cao, Minjia Zhang, Michael D. Bond, Benjamin P. Wood

A Deployable Sampling Strategy for Data Race Detection by Yan Cai1, Jian Zhang, Lingwei Cao, and Jian Liu