Bachelor Thesis

Robin Willenbrock

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

June 30, 2024

supervised by:
Ulrike Engeln

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

Contents

List of Figures

1 Introduction

2 Background

2.1 Interrupt-Driven Systems

An interrupt-driven system is an architecture where the flow of execution is changed by unpredictable events in the system, also known as interrupts. Interrupts can be caused by hardware devices, software conditions, or external signals forcing the processor to suspend the current task to execute an interrupt handler or interrupt service routine (ISR). Interrupt-driven systems are used in real-time operating systems, embedded systems, and generally in systems where timely responses are necessary [**?**].
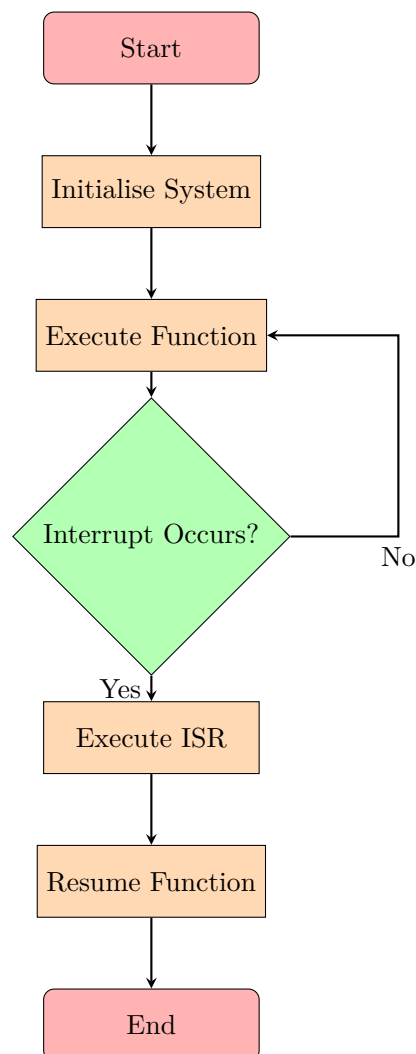

Figure 2.1: Flow-Chart des interruptgesteuerten Systems

In Figure 2.1, a basic execution flow of a simple interrupt-driven system is displayed. The system executes a function as long as no interrupt occurs. When an interrupt occurs, it switches to the ISR, executes it, and then resumes the function executed before the interrupt happened.

The management of the interrupts to maintain the fast responsiveness of the system is the most challenging part of an interrupt-driven system. Interrupts occur in unpredictable ways, so you have to consider every possible execution flow. To ensure the execution of critical interrupts, interrupts are often prioritised, so higher priority events can interrupt lower ones and be handled immediately. When handling an interrupt, the current state of the processos is saved, and the context is switched to the ISR [**?**].

The unpredictivity and asynchronous nature of the interrupts present a lot of challenges in designing and implementing an interrupt-driven system. One of the biggest challenges is the correct handling of high-priority interrupts without delaying them substantially. Which needs a sophisticated scheduling and prioritisation mechanism. The execution of the main programme and ISR needs to be handled properly to ensure data tegrity. Furthermore, handling context switches, preserving system state, and avoiding deadlocks all contribute to the development of an interrupt-driven system.

2.2 Shared Resources

Shared resources, often referred to as shared memory or shared variables, are data that can be accessed simultaneously by multiple threads or processes. Proper management of these resources is crucial because improper handling can lead to issues like data races, deadlocks, and other synchronisation problems. In interrupt-driven systems, shared resources often involve variables or data structures that are accessed by both the main programme and ISRs. Proper management of shared resources is critical to ensuring data consistency and avoiding conflicts [**?**]. Proper management of shared resources involves the use of synchronisation mechanisms to coordinate access and ensure data consistency. Mutexes, semaphores, and condition variables are common tools used to control access to shared resources. Mutexes provide mutual exclusion, ensuring that only one thread can access the resource at a time. Semaphores can limit the number of threads accessing the resource simultaneously. Condition variables allow threads to wait for certain conditions to be met before proceeding, facilitating complex synchronisation scenarios [**?**]. In interrupt driven software, the synchronisation of the shared resources often implies disabling-enabling interrupts [**?**]. Analysing the management of the shared resources is a large part of the data race analysis, which is further explained later.

2.3 Reduced Inter-Procedural Control Flow Graphs (RICFG)

Control Flow Graphs (CFG) are representations of all possible paths through a programme or a function during its execution. An Inter-Procedural Control Flow Graph (ICFG) adds possible edges between multiple programmes or functions to also show possible control flows between those. A Reduced Inter-Procedural Control Flow Graph

(RICFG) is an optimised version of the ICFG that simplifies the graph to only the necessary information needed for the analysis [**?**].
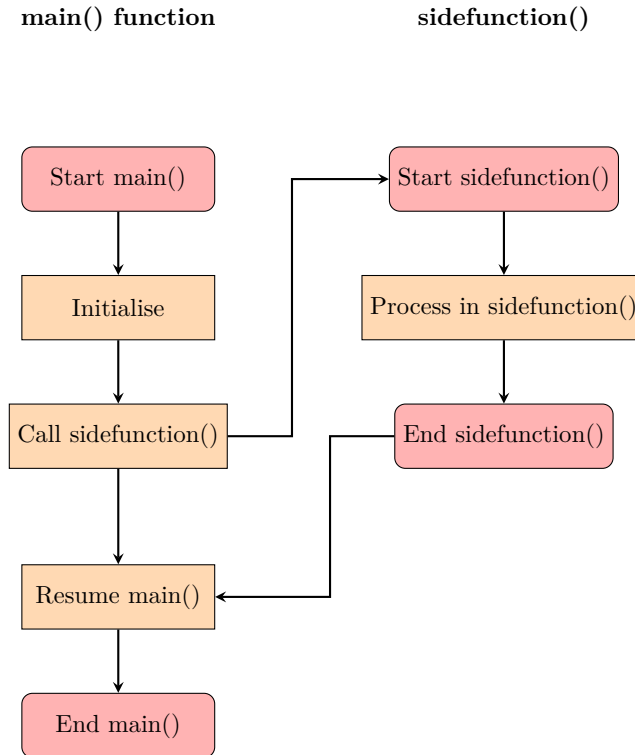

Figure 2.2: Example of Inter-Procedural Control Flow Graph

In Figure 2.2, a simple ICFG is shown. There are two separate linear control flow graphs where the main function calls the sidefunction in its execution. To interpret the flow of the programme correctly, you need to consider the execution of sidefunction() and where it's called. The ICFG combines the two separate CFGs to ensure correct analysis.

There are multiple techniques to reduce the graph, such as node merging, edge contraction, and the elimination of non-important nodes, without losing any information required for the analysis and reducing the complexity of the RICFG. The reduction of the ICFG makes the analysis of large and complex software a lot more efficient. By minimising the amount of data while retaining enough detail, RICFGs are great for static analysis of data races [**?**].

Node merging is combining nodes that represent redundant control flow paths to reduce the number of nodes in the graph. Edge contraction is simplifying the graph by reducing the number of edges between nodes. It collapses edges, that do not significantly affect the control flow of the graph. [**?**] The elimination of nodes is the main tool used in this work to reduce the CFG. Eliminating nodes that do not carry any essential information for the applied data analysis significantly reduces the amount of data the

algorithm has to analyze. Overall, these techniques enhance the scalability of static analysis and make it more practical to analyse more complex data [**?**].

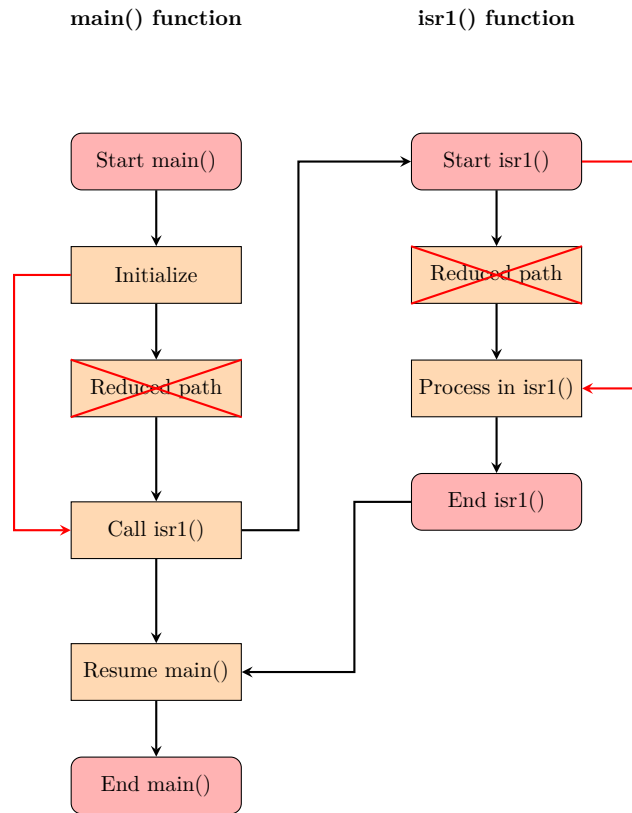**main() function**   **isr1() function**

Figure 2.3: Example of Reduced Inter-Procedural Control Flow Graph

Figure 2.3 shows an example of a simple reduction by eliminating nodes that do not carry any important information for the analysis the RICFG is used for.

2.4 Data Races

A data race occurs when two or more functions or threads access a shared resource concurrently, without being ordered by a happens-before relationship, and one of those accesses is a write operation [**?**]. This can lead to unpredictable behaviour and errors in the system, which makes the detection of data races a critical aspect of concurrent programmes. Without proper synchronisation, a system with multiple threads or functions that use shared data will lead to data races. The outcome of a programme with data races is non-deterministic [**?**]. The order of execution of operations can vary, which may result in the generation of bugs that are not reproducible or difficult to reproduce.

Algorithm 1: Data Race Example

   Data: long shared1;

1 Function `main` *():*

2   **Variables**:

3   unsigned char tmp;

4   **Code**:

5   tmp ← shared1;

6 Function `isr1` *():*

7   **Code**:

8   idlerun();

9   shared1 ← 1;

10   idlerun();

Figure 2.4: Simple Example of a Data Race

In Figure 2.4, an example of a simple data race is shown. A global variable shared1 is initiated and accessed in two different functions main() and isr1 (). Since there are no synchronisation tools used and the operation in isr1 is a write, there is a data race between line 5 and line 9.

2.4.1 Detection Techniques

Data race detection can be approached by two different analytical methods. Each of those methods provides benefits and challenges.

Static Data Race Detection [?]

Advantages:

- Comprehensivness: Static analysis inspects the code without executing the programme by analysing every possible execution path and interactions that could lead to data races.

- Early Detection: Since static analysis does not require execution, it can analyse the code in the development phase, allowing the developer to find issues without deployment.

Disadvantages:

- False Positives/Negatives: Static analysis reports all data races that fall under certain conditions. Some of these data races could be very unlikely or even impossible at runtime. On the other hand, due to the approximations and assumptions necessary for tractability, it may miss some races.

- Complexity in Handling Dynamic Behaviour: Dynamic behaviours such as pointers or recursion can be challenging to analyse for static approaches, leading to incomplete or inaccurate results.

Dynamic Data Race Detection [?]

Advantages:

- Precision: Dynamic analysis tools monitor the actual execution of a programme, identifying data races in real-time. Which results in reducing the number of false positives.

- Context-Sensitive Detection: By analysing the actual runtime behaviour, dynamic analysis can understand the context of operations, leading to more accurate detection.

Disadvantages:

- Performance Overhead: The analysis in runtime can slow down the application significantly.

- Coverage: The effectiveness is heavily dependent on the execution path triggered during the tests. If certain parts of the programme are not passed through in the execution run, they are not analysed.

Both static and dynamic analysis are crucial for a complete analysis of a code. They complement each other's limitations. A combination of both is the best approach to detecting data races most reliable. However, in this work, I am going to focus on the static analysis of data races.

2.4.2 Strategies for Preventing Data Races

Preventing data races requires careful design and implementation of concurrent programs. One effective strategy is to use proper synchronisation mechanisms, such as mutexes, semaphores, and condition variables, to control access to shared data. These mechanisms ensure that only one thread can access the shared data at a time, preventing conflicting operations. Avoiding shared mutable states is another effective strategy, where threads operate on local copies of data instead of shared data, reducing the potential for conflicts. Designing thread-safe data structures and algorithms that inherently manage concurrent access also helps prevent data races, ensuring reliable and predictable programme behaviour [**?**].

Preventing data races requires careful design and implementation of concurrent programs. Effective strategies for general prevention of data races are synchronisation mechanisms such as mutexes, semaphors, and condition variables, which control access to shared data. Those mechanisms ensure that only one thread can access the shared resource at a time [**?**]. Since I am focusing on data races in interrupt driven systems,

the main tool to prevent data races is to disable ISRs, which access shared resources in critical areas.

Algorithm 2: Enable/Disable ISR Call Example

    Data: long shared1

1 Function *main()*:

2   **Variables**:

3   unsigned char tmp;

4   **Code**:

5   disable_isr(1);

6   tmp ← shared1;

7   enable_isr(1);

8 Function *isr1()*:

9   **Code**:

10   idlerun();

11   shared1 ← 1;

12   idlerun();

13 Function *isr2()*:

14   **Code**:

15   idlerun();

16   int variable1 = 1;

17   idlerun();

Figure 2.5: Example of a Data Race with Enable/Disable ISR Calls

Figure 2.5 is an example of a disable ISR call that leads to the save access of the shared data. The main function and isr1 both access the shared resource shared1. Since the read operation in line 6 of the main function is safely accessed by disabling isr1 in line 5 and enabling it in line 7, a possible data race is prevented.

2.5 Static Detection of Data Races in Interrupt-Driven Systems

The asynchronous nature and concurrent execution of ISRs and the main function introduces significant challenges for data consistency and detecting data races in interrupt-driven systems. Static data race analysis, especially those using RICFGs, are a promising approach to identifying data races without the need for extensive testing and runtime monitoring as in dynamic approaches [**?**].

The static approach involves the construction of an RICFG for the programme, which includes both the main code and ISRs, and capturing the control flow and potential interaction between them. Analysing the RICFG, shows paths where shared resources

are accessed concurrently without proper synchronisation and indicates potential data races. Integrating the static analysis tool with the development process enables continuous detection of data races during software development, improving the reliability and correctness of interrupt-driven systems [**?**].

The methodology for static data race detection in interrupt-driven systems involves the following key steps. First, the RICFGs are constructed for the entire programme, including the main code and the ISRs. This involves analysing the control flow and identifying interactions between the main programme and ISRs. Next, the RICFGs are analysed to find potential data races, focusing on paths where concurrent access of shared data is done without proper synchronization. Finally, the developer can use the analysis results to address identified data races in early development processes [**?**].

Algorithm 3: Static Race Detection

   Input: RICFGs of P
   Output: potential racing pairs (PR)

1 **for** *each* $< G_i; G_j >$ *in RICFGs* **do**
2   **for** *each* $sv_i \in G_i$ **do**
3   **for** *each* $sv_j \in G_j$ **do**
4   **if** $sv_i.V == sv_j.V$ *and* $(sv_i.A == W$ *or* $sv_j.A == W)$ *and* $G_i.pri < G_j.pri$ *and* $INTB.get(svi).contains(Gj)$ **then**
5 $PR = PR \cup \{< sv_i, sv_j >\};$

Figure 2.6: Static Race Detection Approach by [**?**]

The approach by Wang et al. shows a computation of potential data races using RICFGs. By running a depth-first search on the RICFGs it finds the interrupt status of every instruction. If there is a shared resource in both of the analysed RICFGs, at least one of them is a write operation, and the two functions differ in their priority. While the interrupt in this pair is enabled, the two accesses are a potential data race [**?**]. In the following, I am going to introduce you to the implementation of my static analysis programme based on the static race detection approach of Wang et al..

3 Implementation

In the following I will provide an indepth explanation of my implemenation. For the generation of the input I used GCC. The command `gcc -fdump-tree-cfg` provides a cfg-file with all important informations for the intended data race analysis. I have split the explanation of the implementation into the initialization of the basic block class, the parsing of the input, the actual data race analysis and the filter of false positives found in data race analysis.

Class BasicBlock

Algorithm 4: BasicBlock Class Definition

---
1 Class BasicBlock:

2   **Def** __init__(self, function_name, number, shared_resources, successors, enable_disable_calls, code):

3   self.function_name ← function_name

4   self.number ← number

5   self.shared_resources ← shared_resources

6   self.successors ← successors

7   self.enable_disable_calls ← enable_disable_calls

8   self.code ← code

The class `BasicBlock` is a display of all the information nessessary for the data race analysis found in the input. Those informations include the following attributes:

- `function_name`: The function name to which the basic block belongs.

- `number`: The number of the basic block.

- `shared_resources`: All accesses of shared resources within the basic block. The access type (read/write) and also the line number of such calls are saved.

- `successors`: A list of all the successors of each basic block. Important to build all possible paths through the CFG.

- `enable_disable_calls`: All calls that disable or enable an ISR within this basic block and also with the corrsponding line number of those calls to ensure correct order.

- `code`: The lines of code of the block stored with their line number.

Parsing

Algorithm 5: Parse Basic Blocks

Data: file_path, shared_resource_names

Result: Dictionary of BasicBlocks mapping (function name, block number) to
   BasicBlock instances

1 blocks ← empty dictionary

2 current_function ← None

Input: Path to source file, list of shared resource names

Output: Dictionary of basic blocks with details

---

3 for *each line in file at file_path* **do**

4 │ line ← line.strip()

5 │ Increment line_number

6 │ **if** *line starts new function* **then**

7 │ │ **if** *current block is open* **then**

8 │ │ │ Save current block to blocks

9 │ │ **end**

10 │ │ Update current_function and reset current block details

11 │ **end**

12 │ **else if** *line starts new basic block* **then**

13 │ │ **if** *current block is open* **then**

14 │ │ │ Save current block to blocks

15 │ │ **end**

16 │ │ Initialize new block details

17 │ **end**

18 │ Update shared_resources and enable_disable_calls based on line content

19 end

20 if *current block is open at end of file* **then**

21 │ Save last block to blocks

22 end

23 foreach *line indicating successor blocks* **do**

24 │ Parse successor details and update block information in blocks

25 end

26 return *blocks*

The function `parse_basic_blocks` starts with the initialisation of a dictionary `blocks` and a variable `current_function`. It opens the file and reads its lines into a list `lines`. These variables are initialised to collect information on basic blocks:

- `bb_num`: Number of the current basic block.

- `shared_resources`: List of shared resources in the current block.

- `enable_disable_calls`: List of ISR activation/deactivation calls.

- `code_lines`: The lines of code of the current block.

- `line_number`: The current line number in the file.

The loop runs through each line of the file, removes leading and trailing spaces and increments the line number. If a new function is recognised (by the pattern `;; Function ... ()`), the current block is saved and the current function is updated.

If a new basic block number is detected (`<bb +>:`), the previous block is saved and the information for the new block is initialised.

This section searches each line for shared resources and ISR activation/deactivation calls. Resources and calls found are added to the corresponding list and the current line is added to `code_lines`.

At the end of the loop, the last basic block is saved if there is still an open block.

In a second pass through the lines, the successor blocks for each basic block are identified and linked accordingly. Finally, the function returns the `blocks` dictionary.

Algorithm 6: Track ISR Status

Data: blocks
Result: List indicating ISR status initialized to zero
Input: Dictionary of BasicBlocks
Output: List of zeros representing the status of each ISR

1   isr_count ← count unique ISRs in function names from blocks
2   **return** *list initialized to zero of length isr_count*

The function `track_isr_status` initialises the tracking of the ISR status. It returns a list of zeros whose length corresponds to the number of unique ISRs in the programme. This list is used to track the activation/deactivation status of each ISR.

Algorithm 7: Extract ISR Index

Data: function_name
Result: Zero-based index of the ISR or None if no ISR is found
Input: Function name as a string
Output: Integer index or None

1   match ← search for pattern 'isr[_]?(+)' in function_name
2   **if** *match is found* **then**
3 | **return** *integer value of the first group in match minus one*
4   **else**
5 | **return** *None*
6   **end**

The function `extract_isr_index` extracts the index of an ISR from a function name that follows the pattern `isr_+`. It returns the index of the ISR reduced by one to enable zero-based indexing.

Data Race Detection

Algorithm 8: Detect Data Races

Data: blocks
Result: List of potential data races identified
Input: Dictionary of BasicBlocks
Output: List of potential data races

---

1 potential_data_races ← empty list
2 resource_accesses ← initialize as a default dictionary to list
3 isr_enabling_map ← initialize as a default dictionary to set

4 **foreach** *block in blocks* **do**
5   **foreach** *call, line_number in block.enable_disable_calls* **do**
6   **if** *call contains 'enable_isr'* **then**
7   isr_idx_match ← search for pattern '+' in call
8   **if** *isr_idx_match is found* **then**
9   enabled_isr_idx ← integer value of the first group in
                    isr_idx_match minus one
10   enabler_isr ← block.function_name
11   isr_enabling_map[enabler_isr].add(enabled_isr_idx)
12   **end**
13   **end**
14   **end**
15 **end**

The first part of the function `detect_data_races` runs through all blocks and collects information about ISR activations. This information is stored in the `isr_enabling_map`, which tracks the ISR activation relationships.

Algorithm 9: Process Block

Data: block, current_isr_status

Result: Updated ISR status and recorded resource accesses

Input: A code block and the current ISR status as a list

Output: Updated current ISR status and appended resource accesses to a global list

---

1 **foreach** *line, line_number in block.code* **do**
2   **if** *line contains 'enable_isr' or 'disable_isr'* **then**
3   isr_idx_match ← search for pattern '+' in line
4   **if** *isr_idx_match is found* **then**
5   isr_idx ← integer value of the first group in isr_idx_match minus one
6   **if** *"disable_isr" is in line* **then**
7   **if** *0 ≤ isr_idx < length of current_isr_status* **then**
8   current_isr_status[isr_idx] ← 1
9   **end**
10   **else**
11   **if** *0 ≤ isr_idx < length of current_isr_status* **then**
12   current_isr_status[isr_idx] ← 0
13   **end**
14   **end**
15   **end**
16   **end**
17   **foreach** *resource_name, access_type, res_line_number in block.shared_resources* **do**
18   **if** *res_line_number == line_number* **then**
19   resource_accesses[resource_name].append((block.function_name, block.number, access_type, line_number, current_isr_status.copy()))
20   **end**
21   **end**
22 **end**

The `process_block` function processes a single block and updates the current ISR status based on `enable_isr` and `disable_isr` calls. Accesses to shared resources are stored in `resource_accesses`.

Algorithm 10: Depth-First Search (DFS) for Blocks

Data: block, visited_blocks, current_isr_status, path
Result: Recursive processing of blocks and their successors
Input: A block, set of visited blocks, current ISR status, and path tracing the blocks visited
Output: Processed blocks with updated ISR statuses and path

---

1 if _(block.function_name, block.number) is in visited_blocks_ **then**
2 | **return**
3 end

4 visited_blocks.add((block.function_name, block.number))
5 path.append((block.function_name, block.number))

6 process_block(block, current_isr_status)

7 if _block has successors_ **then**
8 | **foreach** _successor in block.successors_ **do**
9 | | dfs(successor, set(visited_blocks), current_isr_status.copy(), path.copy())
10 | **end**
11 end

The function `dfs` (Depth-First Search) runs through the control flow graph and tracks the ISR status. Each block is processed and the ISR status and accesses to resources are updated.

Algorithm 11: Initial Processing and Recursive Depth-First Search

---

Data: blocks (a dictionary of blocks indexed by function name and block number)
Result: Executes process block and DFS on selected blocks and their successors

---

1 foreach _(func_name, bb_num) and block in blocks_ **do**
2 | **if** _bb_num = 2_ **then**
3 | | initial_isr_status ← track_isr_status(blocks)
4 | | process_block(block, initial_isr_status)
5 | | **foreach** _successor in block.successors_ **do**
6 | | | dfs(successor, {}, initial_isr_status.copy(), [(func_name, bb_num)])
7 | | **end**
8 | **end**
9 end

This section initialises the depth search for basic blocks with the number 2 and starts processing and running through the successor blocks.

Algorithm 12: Check for Data Races

Result: Update global list of potential data races based on access patterns
Output: Updated list of potential data races with detailed conflict info

---

1 **foreach** *resource, accesses in resource_accesses* **do**
2   **for** *i ← 0* **to** *length(accesses)-1* **do**
3   func1, bb_num1, access_type1, line_number1, isr_status1 ← accesses[i]
4   **for** *j ← i+1* **to** *length(accesses)-1* **do**
5   func2, bb_num2, access_type2, line_number2, isr_status2 ← accesses[j]
6   **if** *func1 ≠ func2 and (access_type1 = "write" or access_type2 = "write")* **then**
7   potential_data_races.append((resource, (func1, bb_num1, access_type1, line_number1, isr_status1), (func2, bb_num2, access_type2, line_number2, isr_status2)))
8   **end**
9   **end**
10   **end**
11 **end**

The function `check_for_data_races` identifies potential data races by comparing pairs of resource accesses. If two accesses to the same resource originate from different functions and at least one of them is a write access, a potential data race is identified.

Calling `check_for_data_races` performs a check of the potential data races.

Filter Possible Data Races

Algorithm 13: Filter Data Races

Data: potential_data_races

Result: Filtered list of data races

1 Function `filter_data_races`(*potential_data_races*):

2   filtered_data_races ← empty list

3   **foreach** *resource, access1, access2 in potential_data_races* **do**

4   Decompose access1 into func1, bb_num1, access_type1, line_number1, isr_status1

5   Decompose access2 into func2, bb_num2, access_type2, line_number2, isr_status2

6   relevant_isr_disabled1 ← `is_isr_disabled`(*isr_status1, func2*) and not `is_isr_enabled_by_another`(*isr_status1, func2*)

7   relevant_isr_disabled2 ← `is_isr_disabled`(*isr_status2, func1*) and not `is_isr_enabled_by_another`(*isr_status2, func1*)

8   **if** *not (relevant_isr_disabled1 or relevant_isr_disabled2)* **then**

9   filtered_data_races.append((resource, access1, access2))

10   **end**

11   **end**

12   **return** *filtered_data_races*

13 end

14 Function `is_isr_disabled`(*isr_status, func_name*):

15   isr_idx ← extract_isr_index(func_name)

16   **if** *isr_idx is not None and isr_idx < length of isr_status* **then**

17   **return** *isr_status[isr_idx] == 1*

             `// 1 indicates disabled`

18   **end**

19   **return** *False*

20 end

21 Function `is_isr_enabled_by_another`(*isr_status, func_name*):

22   isr_idx ← extract_isr_index(func_name)

23   **if** *isr_idx is not None* **then**

24   **foreach** *enabler_isr, enabled_isrs in isr_enabling_map* **do**

25   enabler_idx ← extract_isr_index(enabler_isr)

26   **if** *enabler_idx is not None and not* `is_isr_disabled`(*isr_status, enabler_isr*) **then**

27   **if** *isr_idx in enabled_isrs* **then**

28   **return** *True*

29   **end**

30   **end**

31   **end**

32   **end**

33   **return** *False*

34 end

The function `filter_data_races` filters false positives from the list of potential data races. It checks the ISR status during resource accesses and only confirms data races if relevant ISRs were not deactivated at access time or activated by another function. At the end, `detect_data_races` returns the filtered data races.

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

Bibliography

[1] Wang, Y., Gao, F., Wang, L., Yu, T., Zhao, J., & Li, X. (2020). Automatic Detection, Validation, and Repair of Race Conditions in Interrupt-Driven Embedded Software. IEEE Transactions on Software Engineering.

[2] Engler, D., & Ashcraft, K. (2003). RacerX: Effective, Static Detection of Race Conditions and Deadlocks. ACM SIGOPS Operating Systems Review.

[3] Nikita Chopra, Rekha Pai, and Deepak D'Souza (2019). Data Races and Static Analysis for Interrupt-Driven Kernels

[4] Flanagan, C., & Freund, S. N. (2009). FastTrack: Efficient and Precise Dynamic Race Detection. ACM SIGPLAN Notices.

[5] R. Chen, Xiangying Guo, Y. Duan, B. Gu, Mengfei Yang (2011). Static Data Race Detection for Interrupt-Driven Embedded Software.

[6] Muchnick, S. S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann.

[7] Adve, S. V., & Gharachorloo, K. (1996). Shared Memory Consistency Models: A Tutorial. IEEE Computer.

[8] Herlihy, M., & Shavit, N. (2008). The Art of Multiprocessor Programming. Morgan Kaufmann.

Attachments

```
1   class BasicBlock:
2   def __init__(function_name, number, shared_resources=[],
        successors=[], enable_disable_calls=[], code=[]):
3   self.function_name = function_name
4   self.number = number
5   self.shared_resources = shared_resources
6   self.successors = successors
7   self.enable_disable_calls = enable_disable_calls
8   self.code = code
9
10  def __repr__():
11  return ("BasicBlock(function_name={}, number={}, shared_resources
        ={}, "
12  "successors={}, enable_disable_calls={}, code={})".format(
13  self.function_name, self.number, self.shared_resources,
14  [succ.number for succ in self.successors], self.
        enable_disable_calls,
15  ' '.join(self.code)))
16
17  def parse_basic_blocks(file_path, shared_resource_names):
18  blocks = {}
19  current_function = None
20
21  with open(file_path, 'r') as file:
22  lines = file.readlines()
23
24  bb_num = None
25  shared_resources = []
26  enable_disable_calls = []
27  code_lines = []
28  line_number = 0
29
30  for line in lines:
31  line = line.strip()
32  line_number += 1
33
34  func_match = re.match(r';; Function (.+?) \(', line)
35  if func_match:
36  if bb_num is not None and current_function is not None:
37  blocks[(current_function, bb_num)] = BasicBlock(
38  current_function, bb_num, shared_resources, [],
        enable_disable_calls, code_lines)
39  current_function = func_match.group(1)
40  bb_num = None
41  continue
42
43  bb_match = re.match(r'<bb (\d+)>:', line)
44  if bb_match:
45  if bb_num is not None and current_function is not None:
46  blocks[(current_function, bb_num)] = BasicBlock(
```

```
47          current_function , bb_num , shared_resources , [] ,
                enable_disable_calls , code_lines )
48          bb_num = int( bb_match.group (1))
49          shared_resources = []
50          enable_disable_calls = []
51          code_lines = []
52
53          for resource_name in shared_resource_names :
54          if re.search(fr'\b{resource_name}\b', line ):
55          if re.search(fr'\b{resource_name}\b\s*=', line ):
56          shared_resources.append (( resource_name , 'write', line_number ))
57          else :
58          shared_resources.append (( resource_name , 'read', line_number ))
59
60          if 'enable_isr' in line or 'disable_isr' in line :
61          enable_disable_calls.append (( line.strip() , line_number ))
62
63          code_lines.append (( line , line_number ))
64
65          if bb_num is not None and current_function is not None :
66          blocks [( current_function , bb_num )] = BasicBlock (
67          current_function , bb_num , shared_resources , [] ,
                enable_disable_calls , code_lines )
68
69          current_function = None
70          bb_num = None
71          for line in lines :
72          line = line.strip()
73
74          func_match = re.match(r';; Function (.+?) \(', line )
75          if func_match :
76          current_function = func_match.group (1)
77          bb_num = None
78          continue
79
80          if 'succs' in line :
81          succ_match = re.match(r';; (\d+) succs \{(.+?)\}', line )
82          if succ_match :
83          bb_num = int( succ_match.group (1))
84          succ_list = [int( succ.strip()) for succ in succ_match.group (2).
                split ()]
85          if ( current_function , bb_num ) in blocks :
86          blocks [( current_function , bb_num )]. successors = [
87          blocks [( current_function , succ )] for succ in succ_list if (
                current_function , succ ) in blocks]
88
89          return blocks
90
91          def track_isr_status ( blocks ):
92          isr_count = len(set( block.function_name for block in blocks.
                values () if re.search(r'isr[_]?\d+', block.function_name )))
93          return [0] * isr_count
94
95          def extract_isr_index ( function_name ):
```

```
96              match = re.search(r'isr[_]?(\d+)', function_name)
97              if match:
98              return int(match.group(1)) - 1
99              return None
100
101         def detect_data_races(blocks):
102              potential_data_races = []
103              resource_accesses = defaultdict(list)
104              isr_enabling_map = defaultdict(set)
105
106              for block in blocks.values():
107              for call, line_number in block.enable_disable_calls:
108              if 'enable_isr' in call:
109              isr_idx_match = re.search(r'\((\d+)\)', call)
110              if isr_idx_match:
111              enabled_isr_idx = int(isr_idx_match.group(1)) - 1
112              enabler_isr = block.function_name
113              isr_enabling_map[enabler_isr].add(enabled_isr_idx)
114
115         def process_block(block, current_isr_status):
116              for line, line_number in block.code:
117              if 'enable_isr' in line or 'disable_isr' in line:
118              isr_idx_match = re.search(r'\((\d+)\)', line)
119              if isr_idx_match:
120              isr_idx = int(isr_idx_match.group(1)) - 1
121              if "disable_isr" in line:
122              if 0 <= isr_idx < len(current_isr_status):
123              current_isr_status[isr_idx] = 1
124              elif "enable_isr" in line:
125              if 0 <= isr_idx < len(current_isr_status):
126              current_isr_status[isr_idx] = 0
127
128
129              for resource_name, access_type, res_line_number in block.
                    shared_resources:
130              if res_line_number == line_number:
131              resource_accesses[resource_name].append((block.function_name,
                    block.number, access_type, line_number, current_isr_status.
                    copy()))
132
133         def dfs(block, visited_blocks, current_isr_status, path):
134              if (block.function_name, block.number) in visited_blocks:
135              return
136              visited_blocks.add((block.function_name, block.number))
137              path.append((block.function_name, block.number))
138
139              process_block(block, current_isr_status)
140
141              if not block.successors:
142              pass
143              else:
144              for successor in block.successors:
145              dfs(successor, set(visited_blocks), current_isr_status.copy(),
                    path.copy())
```

```
146
147
148         for (func_name , bb_num), block in blocks.items():
149         if bb_num == 2:
150         initial_isr_status = track_isr_status(blocks).copy()
151         process_block(block , initial_isr_status)
152         for successor in block.successors:
153         dfs(successor , set(), initial_isr_status.copy(), [(func_name ,
                bb_num)])
154
155
156         def check_for_data_races ():
157         for resource , accesses in resource_accesses.items():
158         for i, (func1 , bb_num1 , access_type1 , line_number1 , isr_status1)
                in enumerate(accesses):
159         for j, (func2 , bb_num2 , access_type2 , line_number2 , isr_status2)
                in enumerate(accesses):
160         if i >= j:
161         continue
162         if func1 != func2 and (access_type1 == "write" or access_type2 ==
                "write"):
163         potential_data_races.append((resource , (func1 , bb_num1 ,
                access_type1 , line_number1 , isr_status1),
164         (func2 , bb_num2 , access_type2 , line_number2 , isr_status2)))
165
166         check_for_data_races ()
167
168
169         def filter_data_races(potential_data_races):
170         filtered_data_races = []
171         for resource , access1 , access2 in potential_data_races:
172         func1 , bb_num1 , access_type1 , line_number1 , isr_status1 = access1
173         func2 , bb_num2 , access_type2 , line_number2 , isr_status2 = access2
174
175         def is_isr_disabled(isr_status , func_name):
176         isr_idx = extract_isr_index(func_name)
177         if isr_idx is not None and isr_idx < len(isr_status):
178         return isr_status[isr_idx] == 1
179         return False
180
181         def is_isr_enabled_by_another(isr_status , func_name):
182         isr_idx = extract_isr_index(func_name)
183         if isr_idx is not None:
184         for enabler_isr , enabled_isrs in isr_enabling_map.items():
185         enabler_idx = extract_isr_index(enabler_isr)
186         if enabler_idx is not None and not is_isr_disabled(isr_status ,
                enabler_isr):
187         if isr_idx in enabled_isrs:
188         return True
189         return False
190
191         relevant_isr_disabled1 = is_isr_disabled(isr_status1 , func2) and
                not is_isr_enabled_by_another(isr_status1 , func2)
```

```python
            relevant_isr_disabled2 = is_isr_disabled(isr_status2, func1) and
                not is_isr_enabled_by_another(isr_status2, func1)

            if not (relevant_isr_disabled1 or relevant_isr_disabled2):
            filtered_data_races.append((resource, access1, access2))

        return filtered_data_races

    filtered_data_races = filter_data_races(potential_data_races)

    return filtered_data_races


    shared_resource_input = input("Enter the names of shared
        resources, separated by commas: ")
    shared_resource_names = [name.strip() for name in
        shared_resource_input.split(',')]

    file_path = input("Enter the file path: ").strip()
    blocks = parse_basic_blocks(file_path, shared_resource_names)

    data_races = detect_data_races(blocks)

    print("Detected Data Races:")
    for resource, access1, access2 in data_races:
    print(f"Resource: {resource}")
    print(f"  Access 1: Function {access1[0]} (BB {access1[1]}), {
        access1[2]}, Line {access1[3]}, ISR Status: {access1[4]}")
    print(f"  Access 2: Function {access2[0]} (BB {access2[1]}), {
        access2[2]}, Line {access2[3]}, ISR Status: {access2[4]}")
    print()
```