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

July 2, 2024

supervised by:
Prof. Dr. Sibylle Schupp
Ulrike Engeln

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

STS
Software
Technology
Systems

# Contents

# List of Figures

# Abbreviations

**ISR** Interrupt Service Routine

**CFG** Control Flow Graph

**ICFG** Inter-Procedural Control Flow Graph

**RICFG** Reduced Inter-Procedural Control Flow Graph

**GCC** GNU Compiler Collection

**DFS** Depth-First Search

**BB** BasicBlock

# 1 Introduction

# 2 Background

In this chapter, I am going to provide a brief overview of all the necessary background information needed to understand static data races in interrupt-driven software using Reduced Inter-Procedural Control Flow Graphs (RICFGs). This information includes basics about interrupt-driven systems, shared resources, RICFGs, and data races as a whole.

## 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 [1].

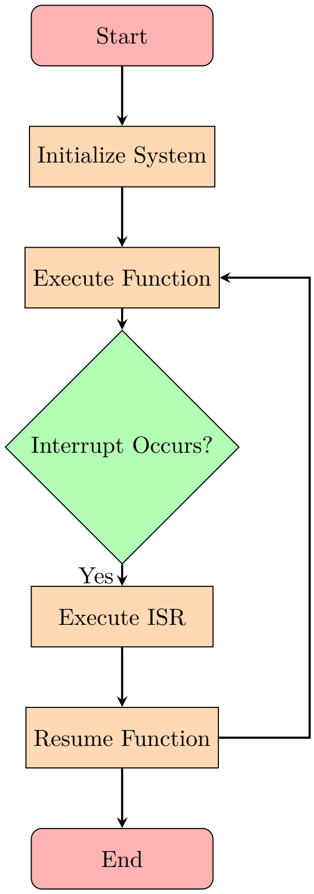


Figure 2.1: Flowchart of the Interrupt-Driven System

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 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 prioritized, so higher-priority events can interrupt lower ones and be handled immediately. When handling an interrupt, the current state of the processor is saved, and the context is switched to the ISR [1].

The unpredictability and asynchronous nature of interrupts present many 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 requires a sophisticated scheduling and prioritization mechanism. The execution of the main program and ISR needs to be handled properly to ensure data integrity.

## 2.2 Shared Resources

Analyzing the management of shared resources is a large part of data race analysis, which is further explained later. The following is a short introduction to shared resources to better understand them in the context of data races.

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 synchronization problems. In interrupt-driven systems, shared resources often involve variables or data structures that are accessed by both the main program and ISRs. Proper management of shared resources is critical to ensuring data consistency and avoiding conflicts [8].

Proper management of shared resources involves the use of synchronization 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 synchronization scenarios [8]. In interrupt-driven software, the synchronization of shared resources often involves disabling and enabling interrupts [3].

## 2.3 Reduced Inter-Procedural Control Flow Graphs

Control Flow Graphs (CFGs) are representations of all possible paths through a program or function during its execution. An Inter-Procedural Control Flow Graphs (ICFGs) adds possible edges between multiple programs or functions to also show possible control flows between those. A RICFG is an optimized version of the ICFG that simplifies the graph to only the necessary information needed for the analysis [2].

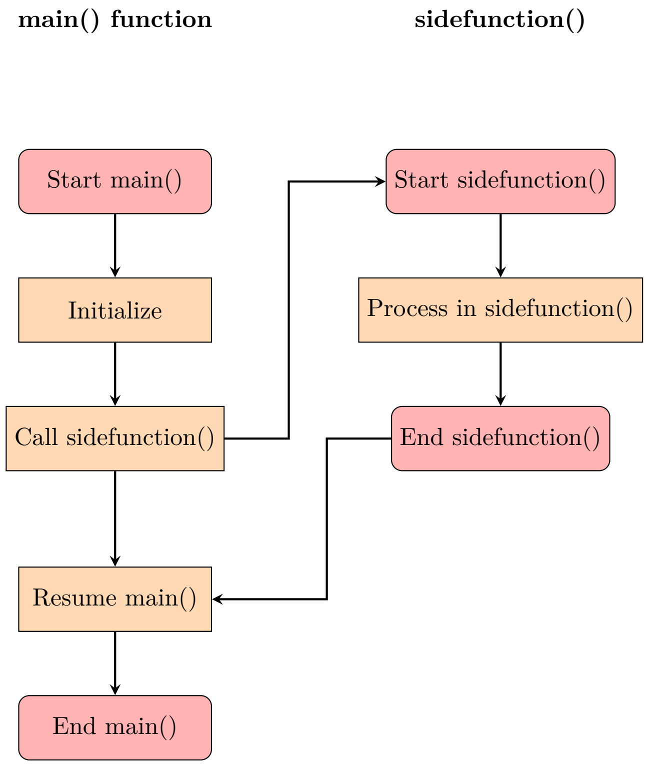


Figure 2.2: Example of an Inter-Procedural Control Flow Graph

In Figure 2.2, a simple ICFG is shown. There are two separate linear control flow graphs where the main function calls the side function in its execution. To interpret the flow of the program 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.

### 2.3.1 Reduction of Control Flow Graphs

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 minimizing the amount of data while retaining enough detail, RICFGs are great for static analysis of data races [1].

Node merging involves combining nodes that represent redundant control flow paths to reduce the number of nodes in the graph. Edge contraction simplifies the graph by reducing the number of edges between nodes. It collapses edges that do not significantly affect the control flow of the graph [6]. 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 analyze more complex data [1].

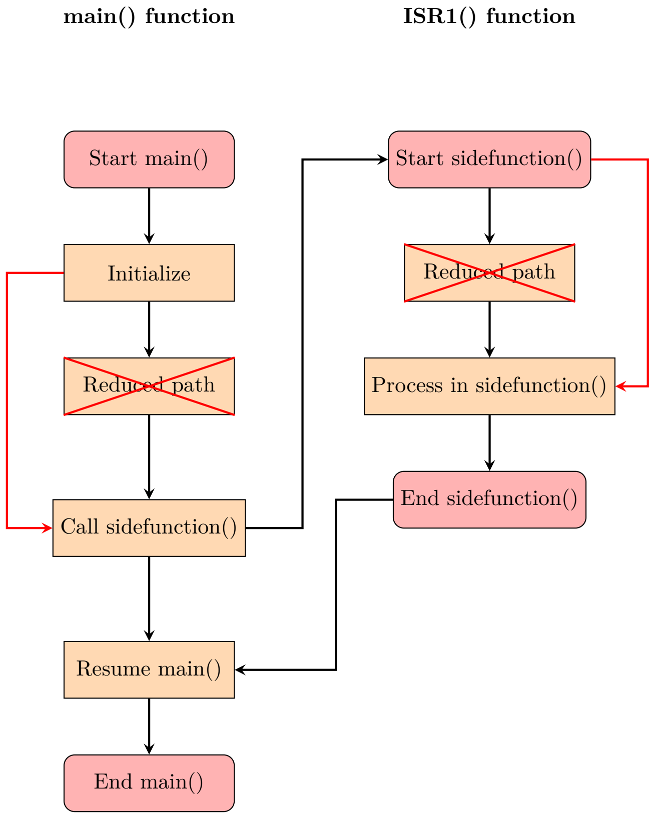


Figure 2.3: Example of a Reduced Inter-Procedural Control Flow Graph

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.

### 2.3.2 Depth-First Search

DFS is an algorithm used to traverse a graph systematically. It begins at a source node and extends its exploration through the connected nodes as far as possible before it backtracks. In an algorithm this can be implemented using a recursive approach. The basic idea is to mark a node when its first discovered and then explore all the adjacent nodes that are not visited before.

```
Function dfs(node, visited):
    visited[node] = True;
    for neighbor in graph[node] do
        if not visited[neighbor] then
            dfs(neighbor, visited);
        end
    end
```

Figure 2.4: Example Implemenation of a Depth-First Search

In Figure 2.4 a simple example of a DFS is shown. The `dfs` function gets called with the root node of a tree and performs the DFS starting from the given node. It recursively calls itself with a new node called `neighbor` and repeats this until all the reachable nodes are marked as visited [9].

## 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 [5]. This can lead to unpredictable behavior and errors in the system, which makes the detection of data races a critical aspect of concurrent programs. Without proper synchronization, a system with multiple threads or functions that use shared data will lead to data races. The outcome of a program with data races is non-deterministic [5]. 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 2.5: 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 synchronization 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 these methods provides benefits and challenges.

#### Static Data Race Detection [1]

**Advantages:**

- Comprehensiveness: Static analysis inspects the code without executing the program by analyzing every possible execution path and interaction that could lead to data races.
- Early Detection: Since static analysis does not require execution, it can analyze 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 Behavior: Dynamic behaviors such as pointers or recursion can be challenging to analyze for static approaches, leading to incomplete or inaccurate results.

#### Dynamic Data Race Detection [4]

**Advantages:**

- Precision: Dynamic analysis tools monitor the actual execution of a program, identifying data races in real-time, which reduces the number of false positives.
- Context-Sensitive Detection: By analyzing the actual runtime behavior, dynamic analysis can understand the context of operations, leading to more accurate detection.

**Disadvantages:**

- Performance Overhead: The analysis at 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 program are not passed through in the execution run, they are not analyzed.

Both static and dynamic analyses are crucial for a complete analysis of code. They complement each other's limitations. A combination of both is the best approach to detecting data races most reliably. 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. Effective strategies for general prevention of data races are synchronization mechanisms such as mutexes, semaphores, and condition variables, which control access to shared data. These mechanisms ensure that only one thread can access the shared resource at a time [8]. 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 2.6: 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 safe 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 introduce significant challenges for data consistency and detecting data races in interrupt-driven systems. Static data race analysis, especially those using RICFGs, is a promising approach to identifying data races without the need for extensive testing and runtime monitoring as in dynamic approaches [1].

The static approach involves the construction of an RICFG for the program, which includes both the main code and ISRs, and capturing the control flow and potential interaction between them. Analyzing the RICFG shows paths where shared resources are accessed concurrently without proper synchronization 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 [1].

The methodology for static data race detection in interrupt-driven systems involves the following key steps. First, the RICFGs are constructed for the entire program, including the main code and the ISRs. This involves analyzing the control flow and identifying interactions between the main program and ISRs. Next, the RICFGs are analyzed to find potential data races, focusing on paths where concurrent access to shared data is done without proper synchronization. Finally, the developer can use the analysis results to address identified data races early in the development process [1].

**Algorithm 3:** Static Race Detection

**Input:** RICFGs of P

**Output:** potential racing pairs (PR)

**for** $each < G_i; G_j > in\ RICFGs$ **do**
    **for** $each\ sv_i \in G_i$ **do**
        **for** $each\ sv_j \in G_j$ **do**
            **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**
                $PR = PR \cup \{< sv_i, sv_j >\}$;

Figure 2.7: Static Race Detection Approach by [1]

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 analyzed 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 [1].

In the following, I am going to introduce you to the implementation of my static analysis program based on the static race detection approach of Wang et al.

# 3 Implementation

In the following, I will provide an in-depth explanation of my implementation. For the generation of the input, I used GNU Compiler Collection (GCC). The command `gcc -fdump-tree-cfg` provides a cfg-file with all the important information 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 filtering of false positives found in data race analysis.

### Class BasicBlock

**Algorithm 4:** BasicBlock Constructor

**Data:** function_name, number, priority, shared_resources, successors, enable_disable_calls, function_calls
**Result:** A BasicBlock object
**Input:** function_name, number, priority, shared_resources (optional), successors (optional), enable_disable_calls (optional), function_calls (optional)
**Output:** A BasicBlock object

**Function** `BasicBlock(`*function_name, number, priority, shared_resources, successors, enable_disable_calls, function_calls*`):`
  self.function_name ← function_name
  self.number ← number
  self.priority ← priority
  self.shared_resources ← shared_resources **if** shared_resources **else** empty list
  self.successors ← successors **if** successors **else** empty list
  self.enable_disable_calls ← enable_disable_calls **if** enable_disable_calls **else** empty list
  self.function_calls ← function_calls **if** function_calls **else** empty list

Figure 3.1: Algorithm: Class BasicBlock

The class `BasicBlock` displays all the information necessary for the data race analysis found in the input. This information includes 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 BasicBlock (BB). The access type (read/write) and the line number of such calls are saved.

- `successors`: A list of all the successors of each basic block. Important for building all possible paths through the CFG.
- `enable_disable_calls`: All calls that disable or enable an ISR within this basic block and also the corresponding line number of those calls to ensure the correct order.
- `function_calls`: The functions that are called within a BB.

Resulting in the following UML class diagramm:

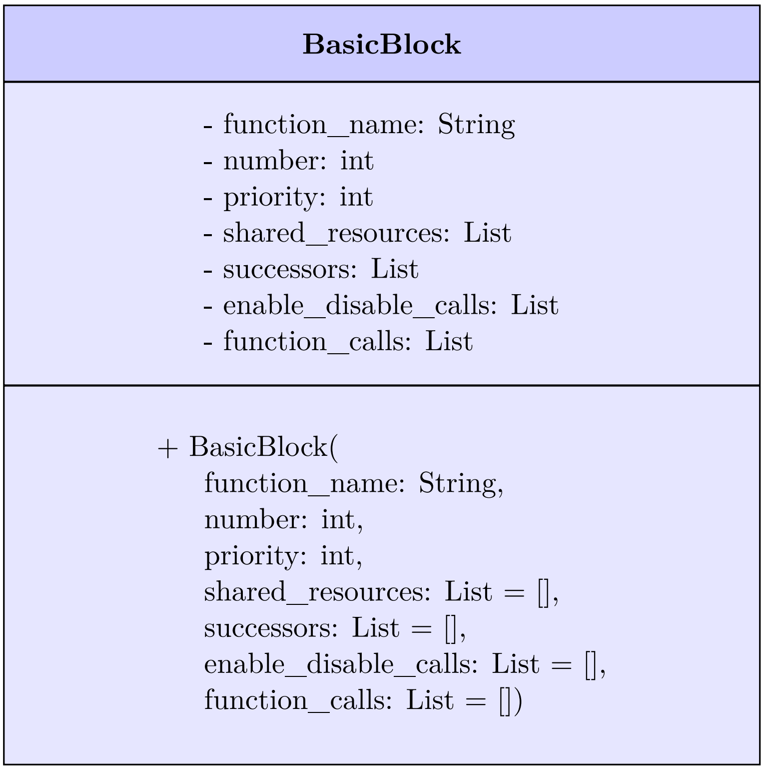


Figure 3.2: UML: Class BasicBlock

### Data Race Detection

The following functions are used to determine all possible data races, which are filtered later in the code. The intention is to find all possible data races to minimize the number of false negatives. Since false positives can be evaluated later by interpreting the output.

**Algorithm 5:** Initialization and ISR Enabling Map

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

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

foreach block in blocks do
    foreach call, line_number in block.enable_disable_calls do
        if call contains 'enable_isr' then
            isr_idx_match ← search for pattern '+' in call
            if isr_idx_match is found then
                enabled_isr_idx ← integer value of the first group in
                  isr_idx_match minus one
                enabler_isr ← block.function_name
                isr_enabling_map[enabler_isr].add(enabled_isr_idx)
            end
        end
    end
end
```

Figure 3.3: Algorithm: Initialization and ISR Enabling Map

The first part of the function `detect_data_races` takes a list of all basic block items as input. It also initializes the empty list of `potential_data_races`, a dictionary for `resource_access`, and a dictionary for the `isr_enabling_map`. `Potential_data_races` and `resource_access` are used later in the code. The main loop of the function iterates through every item in blocks and finds basic blocks with `enable_disable_calls`. If there is an enable call in a block item, the index of the enabled ISR is read, and the basic block is added to the `isr_enabling_map` with the information on which ISR it enables.

**Algorithm 6:** 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

```
foreach line, line_number in block.code do
    if line contains 'enable_isr' or 'disable_isr' then
        isr_idx_match ← search for pattern '+' in line
        if isr_idx_match is found then
            isr_idx ← integer value of the first group in isr_idx_match minus one
            if "disable_isr" is in line then
                if 0 ≤ isr_idx < length of current_isr_status then
                    current_isr_status[isr_idx] ← 1
                end
            else
                if 0 ≤ isr_idx < length of current_isr_status then
                    current_isr_status[isr_idx] ← 0
                end
            end
        end
    end
    foreach resource_name, access_type, res_line_number in
     block.shared_resources do
        if res_line_number == line_number then
            resource_accesses[resource_name].append((block.function_name,
             block.number, access_type, line_number,
             current_isr_status.copy()))
        end
    end
end
```

Figure 3.4: Algorithm: Process Block

This part of the code runs through each line of a basic block to find the current ISR status while resources are accessed. When a resource is found, all the information is added to the `resource_accesses` dictionary, which includes the function name and the block number of the current basic block, as well as the access type, the line number, and the ISR status of the access. All this information is used later for the detection of data races.

**Algorithm 7:** Depth-First Search (DFS) and Initialization

**Data:** blocks
**Result:** Updated block ISR statuses and processed blocks
**Input:** Dictionary of basic blocks
**Output:** Updated block ISR statuses

```
Function dfs(block, visited_blocks, current_isr_status, path):
    if (block.function_name, block.number) in visited_blocks then
        block_isr_statuses[(block.function_name, block.number)] ←
          merge_isr_statuses(block_isr_statuses[(block.function_name,
          block.number)], current_isr_status)
        return
    end
    visited_blocks.add((block.function_name, block.number))
    path.append((block.function_name, block.number))
    block_isr_statuses[(block.function_name, block.number)] ←
      current_isr_status.copy()
    process_block(block, current_isr_status)
    if not block.successors then
        return
    end
    else
        for successor in block.successors do
            dfs(successor, set(visited_blocks), current_isr_status.copy(),
              path.copy())
        end
    end
// Initialization and starting DFS from basic blocks with number 2
for (func_name, bb_num), block in blocks.items() do
    if bb_num == 2 then
        initial_isr_status ← track_isr_status(blocks).copy()
        process_block(block, initial_isr_status)
        for successor in block.successors do
            dfs(successor, set(), initial_isr_status.copy(), [(func_name,
              bb_num)])
        end
    end
end
```

Figure 3.5: Algorithm: DFS and Initialization

The `Depth-First Search (DFS)` function is recursively processing each block in a possible path of the RICFG. The set `visited_blocks` is used to avoid revisiting already visited blocks. If the block is already visited, the ISR status is updated with the stored ISR status for that block using the `merge_isr_statuses` function.

**Algorithm 8:** Merge ISR Statuses

**Data:** isr_status1, isr_status2
**Result:** Merged ISR status list
**Input:** Two lists of ISR statuses
**Output:** List of merged ISR statuses

```
merged_status ← empty list
for isr1, isr2 in zip(isr_status1, isr_status2) do
    merged_status.append(min(isr1, isr2))
end
return merged_status
```

Figure 3.6: Algorithm: Merge ISR Statuses

This function takes the worst case of the most enabled ISRs and uses this for further analysis of the path.

Unvisited blocks get added to the `visited_blocks` set and to the path list. After that, the ISR status gets updated to the current ISR status, and the function `process_block` is called to update the ISR status and track the shared resource accesses.

When the block is processed, the function checks for possible successors and recursively calls itself with the successor and the updated copy of `visited_blocks`, `current_isr_status`, and the path.

The first BB in a function is always the BB with number two in the generated cfg-files. To initialize the DFS, the BB number 2 is processed by the `process_block` function, and after that, the DFS function is called with the successor of the current block.

**Algorithm 9:** Check for Data Races

```
Data: resource_accesses
Result: List of potential data races
Input: Dictionary of resource accesses
Output: List of potential data races
Function check_for_data_races():
    for resource, accesses in resource_accesses.items() do
        for i, (func1, bb_num1, access_type1, line_number1, isr_status1,
          priority1) in enumerate(accesses) do
            for j, (func2, bb_num2, access_type2, line_number2, isr_status2,
              priority2) in enumerate(accesses) do
                if i ≥ j then
                    Continue
                end
                if func1 ≠ func2 and (access_type1 == "write" or access_type2
                  == "write") and priority1 ≠ priority2 then
                    potential_data_races.append((resource, (func1, bb_num1,
                      access_type1, line_number1, isr_status1, priority1), (func2,
                      bb_num2, access_type2, line_number2, isr_status2,
                      priority2)))
                end
            end
        end
    end
// Call the function to check for data races
checkForDataRaces()
```

Figure 3.7: Algorithm: Check for Data Races

The function `check_for_data_races` identifies potential data races by comparing the pairs of data accesses that were initiated earlier. It iterates through all possible tuples of accesses. If a tuple is not within the same function, one of the two accesses is a write operation, and the priorities of both accesses are different, the pair is added to the list of possible data races. All the items in the list fulfill the conditions of a possible data race, which do not include the ISR status tracking. Since the ISR status tracking is the more complex part of the analysis, this makes sure to find all possible data races before filtering to minimize the number of false negatives.

### Filter Possible Data Races

**Algorithm 10:** Filter Data Races

**Data:** potential_data_races, isr_enabling_map
**Result:** Filtered list of data races
**Input:** List of potential data races, ISR enabling map
**Output:** List of filtered data races

```
filtered_data_races ← empty list
seen_races ← empty set
for resource, access1, access2 in potential_data_races do
    func1, bb_num1, access_type1, line_number1, isr_status1, priority1 ←
      access1
    func2, bb_num2, access_type2, line_number2, isr_status2, priority2 ←
      access2
    relevant_isr_disabled1 ← is_isr_disabled(isr_status1, func2) and not
      is_isr_enabled_by_another(isr_status1, func2)
    relevant_isr_disabled2 ← is_isr_disabled(isr_status2, func1) and not
      is_isr_enabled_by_another(isr_status2, func1)
    race_key ← frozenset(((func1, line_number1), (func2, line_number2)))
    if not (relevant_isr_disabled1 or relevant_isr_disabled2) and race_key not
      in seen_races then
        filtered_data_races.append((resource, access1, access2))
        seen_races.add(race_key)
    end
end
return filtered_data_races
```

Figure 3.8: Algorithm: Filter Data Races

The `filter_data_races` function takes the list of possible data races given by the `check_for_data_races` function and filters the racing pairs considering the ISR statuses of the involved ISRs. It takes the two accesses of a potential race and extracts the information that is saved in those accesses. After that, it uses two helper functions to determine the ISR statuses during the access.

**Algorithm 11:** Is ISR Disabled

**Data:** isr_status, func_name
**Result:** Boolean indicating if the ISR is disabled
**Input:** List of ISR statuses, function name as a string
**Output:** Boolean

```
isr_idx ← extract_isr_index(func_name)
if isr_idx is not None and isr_idx < length of isr_status then
    return isr_status[isr_idx] == 1
end
return False
```

Figure 3.9: Algorithm: Is ISR Disabled

The `is_isr_disabled` function checks if the bit corresponding to the ISR in the ISR status array is set to one. If so, the function returns true to the `filter_data_races` function, and if not, it returns false.

**Algorithm 12:** Is ISR Enabled by Another

**Data:** isr_status, func_name, isr_enabling_map
**Result:** Boolean indicating if the ISR is enabled by another function
**Input:** List of ISR statuses, function name as a string, ISR enabling map
**Output:** Boolean

```
isr_idx ← extract_isr_index(func_name)
if isr_idx is not None then
    for enabler_isr, enabled_isrs in isr_enabling_map.items() do
        enabler_idx ← extract_isr_index(enabler_isr)
        if enabler_idx is not None and not is_isr_disabled(isr_status,
          enabler_isr) then
            if isr_idx in enabled_isrs then
                return True
            end
        end
    end
end
return False
```

Figure 3.10: Algorithm: Is ISR Enabled by Another

The `is_isr_enabled_by_another` function looks for possible activations of an ISR by another ISR. The `isr_enabling_map` was initiated and filled with information at the start of the `detect_data_races` function. This information is used in this function to determine if an ISR is enabled by another ISR that is enabled, to correctly handle racing pairs with these conditions.

## Parsing and Helper Functions

In this section the parsing of the input and the helper functions, which are called in the data race analysis, get explained.

```
Algorithm 13: Parse Basic Blocks
Data: file_path, shared_resource_names
Result: blocks, function_blocks
Input: Path to file, List of shared resource names
Output: Dictionary of BasicBlocks, Dictionary of function blocks
Initialize variables
lines ← read lines from file
for line in lines do
    line ← trim(line)
    if match function then
        if bb_num and current_function then
            Add BasicBlock to blocks and function_blocks
        end
        current_function ← extract function name
        bb_num ← None
        Continue;
    end
    if match basic block then
        if bb_num and current_function then
            Add BasicBlock to blocks and function_blocks
        end
        bb_num ← extract basic block number
        Reset lists
    end
    for resource_name in shared_resource_names do
        if resource_name in line then
            if resource_name is written then
                Add write to shared_resources
            end
            else
                Add read to shared_resources
            end
        end
    end
    if line contains enable_isr or disable_isr then
        Add to enable_disable_calls
    end
    if match function call then
        Add function call to function_calls
    end
end
if bb_num and current_function then
    Add BasicBlock to blocks and function_blocks
end
for line in lines do
    line ← trim(line)
    if match function then
        current_function ← extract function name
        bb_num ← None
        Continue;
    end
    if line contains successors then
        Extract successors and update BasicBlock successors
    end
end
return blocks, function_blocks
```

Figure 3.11: Algorithm: Parse Basic Blocks

The `parse_basic_block` function iterates two times through the code to extract all the important information of the code and save it in the BB items. In the first iteration of the code lines, the BBs get initiated with the BB number and the function it relates to. Additionaly the information of shared resources, enable/disable calls of ISRs and function calls within the BB are added. In the second iteration the successors of the BB get added. A second iteration is used to ensure all the BBs items are initialized first to ensure a correct handling of the successors.

**Algorithm 14:** Determine Priority

**Data:** function_name
**Result:** Priority of the function
**Input:** Function name as a string
**Output:** Priority as an integer

match ← search for *'isr_?(+)' in function_name*
**if** *match is found* **then**
  **return** *integer value of matched group* `// Higher priority for lower ISR number`
**end**
**else**
  **return** *infinity* `// Lower priority for non-ISR functions`
**end**

Figure 3.12: Algorithm: Determine Priority

The `determine_priority` function is used to determine the priority of the function which is involved in a possible data race. Since one condition for a data race is two different priorities of the function this is an important check. The priority gets determined in the first place by differentiate between ISR functions and normal functions because ISRs always have a higher priority than non-ISR functions. Non-ISR functions have the priority infinity and ISRs get ordered by the number of it, while lower number ISRs have a higher priority than higher number ones.

**Algorithm 15:** 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

isr_count ← count unique ISR function names in blocks
**return** *list of zeros of length isr_count*

Figure 3.13: Algorithm: Track ISR Status

**Algorithm 16:** Extract ISR Index

**Data:** function_name
**Result:** Index of the ISR
**Input:** Function name as a string
**Output:** ISR index as an integer

```
match ← search for ISR pattern in function_name
if match then
    return integer value of matched group minus one
end
return None
```

Figure 3.14: Algorithm: Track ISR Status

**Algorithm 17:** Propagate Function Calls

**Data:** blocks, function_blocks
**Result:** Updated blocks with propagated function call information
**Input:** Dictionary of BasicBlocks, Dictionary of function blocks
**Output:** Updated BasicBlocks with propagated resources and ISR calls

```
for each function in function_blocks do
    for each block in function's block list do
        for each called_func, line_number in block's function calls do
            if called_func in function_blocks then
                for each called_block in function_blocks[called_func] do
                    block.shared_resources.extend(called_block.shared_resources)
                    block.enable_disable_calls.extend(called_block.enable_disable_calls)
                end
            end
        end
    end
end
```

Figure 3.15: Algorithm: Propagate Function Calls

The `propagate_function_calls` function is handling the case of a function that calls another function. It checks for BBs items with a function call in it and adds the critical parts of the called function to the current BB to simulate a path through the called function and consider the shared resources and enable/disable calls of that function.

# 4 Evaluation

Done:
-Data Race Detection with all necessary criterias for a data race
-Informations of the cfg brought down to the minimum needed to analyze for data races
-Inter-Procedural checks of function calls
-Recursive traversion of the CFG using DFS
-Implementation of an ISR Status Array that updates thorugh the traversal
-Considering ISR enabling ISRs

Not Done/ Future work:
-Pointer analysis
-Unnecessary nodes in CFG are empty but not deleted
-ISR enabling ISR with a depth of more than one are not considered (1 activates 2 activates 3 and 3 has data race)

# 5 Conclusion

# 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.

[9] K. Mehlhorn, P. Sanders (2008). Algorithms and Data Structures The Basic Toolbox.

# Attachments

```
class BasicBlock:
def __init__(function_name, number, shared_resources=[], successors=[], enable_disable_calls=[], code=[]):
self.function_name = function_name
self.number = number
self.shared_resources = shared_resources
self.successors = successors
self.enable_disable_calls = enable_disable_calls
self.code = code

def __repr__():
return ("BasicBlock(function_name={}, number={}, shared_resources={}, "
"successors={}, enable_disable_calls={}, code={})".format(
self.function_name, self.number, self.shared_resources,
[succ.number for succ in self.successors], self.enable_disable_calls,
' '.join(self.code)))

def parse_basic_blocks(file_path, shared_resource_names):
blocks = {}
current_function = None

with open(file_path, 'r') as file:
lines = file.readlines()

bb_num = None
shared_resources = []
enable_disable_calls = []
code_lines = []
line_number = 0

for line in lines:
line = line.strip()
line_number += 1

func_match = re.match(r';; Function (.+?) \(', line)
if func_match:
if bb_num is not None and current_function is not None:
blocks[(current_function, bb_num)] = BasicBlock(
current_function, bb_num, shared_resources, [], enable_disable_calls, code_lines)
current_function = func_match.group(1)
bb_num = None
continue

bb_match = re.match(r'<bb (\d+)>:', line)
if bb_match:
if bb_num is not None and current_function is not None:
blocks[(current_function, bb_num)] = BasicBlock(
current_function, bb_num, shared_resources, [], enable_disable_calls, code_lines)
```

```
bb_num = int(bb_match.group(1))
shared_resources = []
enable_disable_calls = []
code_lines = []

for resource_name in shared_resource_names:
if re.search(fr'\b{resource_name}\b', line):
if re.search(fr'\b{resource_name}\b\s*=', line):
shared_resources.append((resource_name, 'write', line_number))
else:
shared_resources.append((resource_name, 'read', line_number))

if 'enable_isr' in line or 'disable_isr' in line:
enable_disable_calls.append((line.strip(), line_number))

code_lines.append((line, line_number))

if bb_num is not None and current_function is not None:
blocks[(current_function, bb_num)] = BasicBlock(
current_function, bb_num, shared_resources, [], enable_disable_calls, code_lines)

current_function = None
bb_num = None
for line in lines:
line = line.strip()

func_match = re.match(r';; Function (.+?) \(', line)
if func_match:
current_function = func_match.group(1)
bb_num = None
continue

if 'succs' in line:
succ_match = re.match(r';; (\d+) succs \{(.+?)\}', line)
if succ_match:
bb_num = int(succ_match.group(1))
succ_list = [int(succ.strip()) for succ in succ_match.group(2).split()]
if (current_function, bb_num) in blocks:
blocks[(current_function, bb_num)].successors = [
blocks[(current_function, succ)] for succ in succ_list if (current_function, succ) in blocks]

return blocks

def track_isr_status(blocks):
isr_count = len(set(block.function_name for block in blocks.values() if re.search(r'isr[_]?\d+', block.function_name)))
return [0] * isr_count

def extract_isr_index(function_name):
match = re.search(r'isr[_]?(\d+)', function_name)
if match:
```

```
return int(match.group(1)) - 1
return None

def detect_data_races(blocks):
potential_data_races = []
resource_accesses = defaultdict(list)
isr_enabling_map = defaultdict(set)

for block in blocks.values():
for call, line_number in block.enable_disable_calls:
if 'enable_isr' in call:
isr_idx_match = re.search(r'\((\d+)\)', call)
if isr_idx_match:
enabled_isr_idx = int(isr_idx_match.group(1)) - 1
enabler_isr = block.function_name
isr_enabling_map[enabler_isr].add(enabled_isr_idx)

def process_block(block, current_isr_status):
for line, line_number in block.code:
if 'enable_isr' in line or 'disable_isr' in line:
isr_idx_match = re.search(r'\((\d+)\)', line)
if isr_idx_match:
isr_idx = int(isr_idx_match.group(1)) - 1
if "disable_isr" in line:
if 0 <= isr_idx < len(current_isr_status):
current_isr_status[isr_idx] = 1
elif "enable_isr" in line:
if 0 <= isr_idx < len(current_isr_status):
current_isr_status[isr_idx] = 0


for resource_name, access_type, res_line_number in block.shared_resources:
if res_line_number == line_number:
resource_accesses[resource_name].append((block.function_name, block.number, access_type, line_number, current_isr_status.copy()))

def dfs(block, visited_blocks, current_isr_status, path):
if (block.function_name, block.number) in visited_blocks:
return
visited_blocks.add((block.function_name, block.number))
path.append((block.function_name, block.number))

process_block(block, current_isr_status)

if not block.successors:
pass
else:
for successor in block.successors:
dfs(successor, set(visited_blocks), current_isr_status.copy(), path.copy())


for (func_name, bb_num), block in blocks.items():
```

```
if bb_num == 2:
initial_isr_status = track_isr_status(blocks).copy()
process_block(block, initial_isr_status)
for successor in block.successors:
dfs(successor, set(), initial_isr_status.copy(), [(func_name, bb_num)])


def check_for_data_races():
for resource, accesses in resource_accesses.items():
for i, (func1, bb_num1, access_type1, line_number1, isr_status1) in enumerate(accesses):
for j, (func2, bb_num2, access_type2, line_number2, isr_status2) in enumerate(accesses):
if i >= j:
continue
if func1 != func2 and (access_type1 == "write" or access_type2 == "write"):
potential_data_races.append((resource, (func1, bb_num1, access_type1, line_number1, isr_status1),
(func2, bb_num2, access_type2, line_number2, isr_status2)))

check_for_data_races()


def filter_data_races(potential_data_races):
filtered_data_races = []
for resource, access1, access2 in potential_data_races:
func1, bb_num1, access_type1, line_number1, isr_status1 = access1
func2, bb_num2, access_type2, line_number2, isr_status2 = access2

def is_isr_disabled(isr_status, func_name):
isr_idx = extract_isr_index(func_name)
if isr_idx is not None and isr_idx < len(isr_status):
return isr_status[isr_idx] == 1
return False

def is_isr_enabled_by_another(isr_status, func_name):
isr_idx = extract_isr_index(func_name)
if isr_idx is not None:
for enabler_isr, enabled_isrs in isr_enabling_map.items():
enabler_idx = extract_isr_index(enabler_isr)
if enabler_idx is not None and not is_isr_disabled(isr_status, enabler_isr):
if isr_idx in enabled_isrs:
return True
return False

relevant_isr_disabled1 = is_isr_disabled(isr_status1, func2) and not is_isr_enabled_by_another(isr_status1, func2)
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()
```