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 6, 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

List of Algorithms

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 basic block

1 Introduction

Interrupt-driven architectures are crucial in modern software development for timley responses to unpredictable events. These systems often use interrupt service routines (ISRs) to handle such events, to achive the execution of critical tasks with a minimal delay. However, the concurrent nature of ISRs and the executed program can lead to synchronization issues. Data races are one of those critical issues. They can occur when two or more function or ISRs access the same shared resource and potentially lead to inconsistency and unpredictable behavior of the program.

Preventing those problems makes the detection of data races vital for realiable and correct execution of software. Static analysis provides an efficient approach of detecting data races in the development process.

This thesis presents a static data race detection framework specified for interrupt-driven software. The framework uses reduced inter-procedural control flow graphs (RICFGs) to efficiently represent the control flow of the program. By analyzing those graphs, paths where shared resources are accessed concurrently without proper synchonization, are identified and inidicate possible data races. The paths are explored by using a DFS algorithm, which ensures the traversion of every possible path. Additionally, a mechanism to handle enabling and disabling ISRs, a common practice to ensure data consistency in interrupt-driven systems, is implemented.

This thesis is providing a chapter with important backround informations to help understand the overall problem of data races. That chapter is giving a introduction into the topic of interrupt-driven systems, shared resources, reduced inter-procedural control flow graphs and data races.

With the basic knowledge of data races provided, the thesis is going to give an in depth explanation of the implementation of the static analysis framework, using algorithms of the implementation to enhance the understanding of the framework.

The implementation is then evaluated by using public benchmarks and self generated ones to cover a wide spectrum of possible cases. Following the evaluation the results of the framework are discussed, including a overview of the done work and showing possible improvements to the analysis tool that could be done in future work. In the end the thesis is finallized in an conclusion.

2 Background

This chapter provides a brief overview of all the necessary background information needed to understand static data races in interrupt-driven software using reduced interprocedural control flow graphs. 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 determined 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. Interrupt-driven systems are used in real-time operating systems, embedded systems, and generally in systems where timely or event-driven responses are necessary [1].

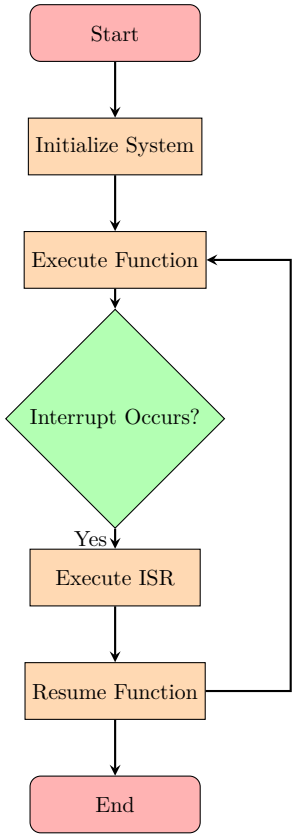

Figure 2.1: Flowchart of the Interrupt-Driven System

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. To conclude in an interrupt-driven system, 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 ensure data consistency and avoiding conflicts [7].

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 [7]. In interrupt-driven software, the synchronization of shared resources often involves disabling interrupts while accessing shared data [3].

2.3 Reduced Inter-Procedural Control Flow Graphs

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.
An inter-procedural control flow graph is a directed graph $G = (V, E)$ where:

- V is the set of vertices. Each vertex represents a basic block (BB) within a procedure or function.
- E is the set of directed edges. Each edge (u, v) represents a possible flow of control from block u to block v. These edges include:
  - Intra-procedural edges represent control flow within a single function.
  - Call edges represent the calling of a function [8].

A RICFG is an optimized version of the ICFG that simplifies the graph to only the necessary information needed for the analysis [2].

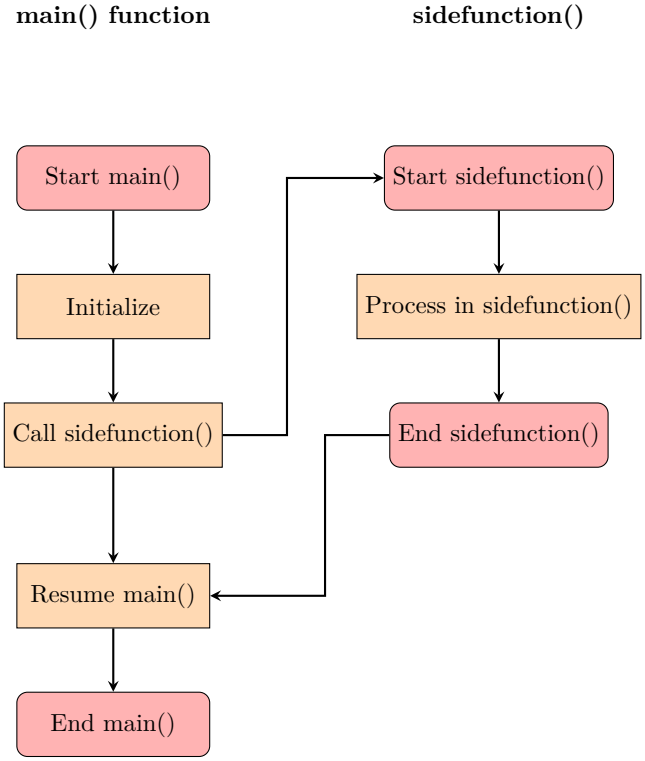

Figure 2.2: Example of an Inter-Procedural Control Flow Graph

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 structures [1].

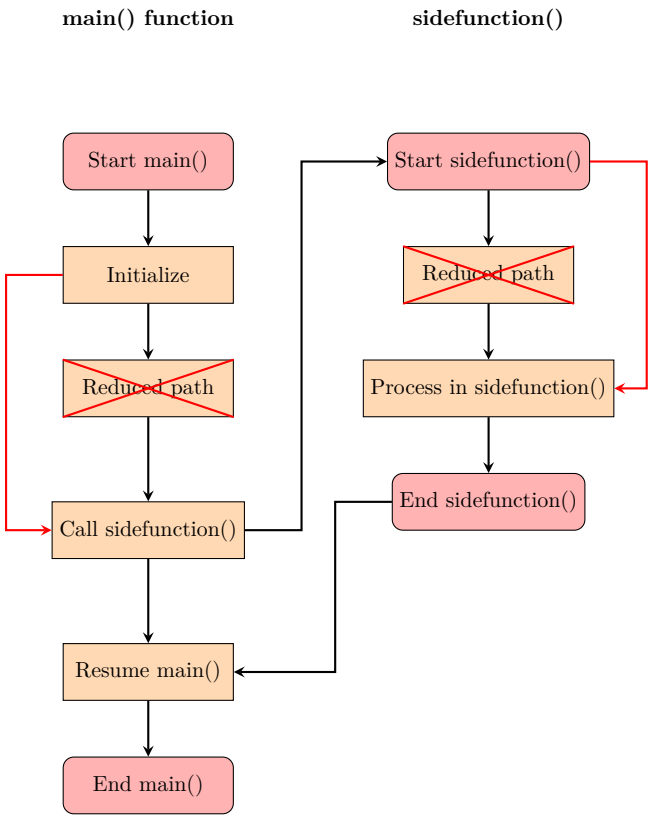

Figure 2.3: Example of a Reduced Inter-Procedural Control Flow Graph

Figure 2.3 shows the example of a simple ICFG with added unneccessary nodes. The resulting RICFG reduces the CFG by eliminating the nodes that do not carry any

important information for the analysis the RICFG is used for. It also adds new edges to skip the deleted nodes.

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.

Algorithm 1: Depth-First Search

```
1 Function dfs(node, visited):
2     visited[node] = True;
3     for neighbor in graph[node] do
4         if not visited[neighbor] then
5             dfs(neighbor, visited);
6         end
7     end
```

In Algorithm 1 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 [8].

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]. A happens-before relationship ensures that, if there are two operations A and B and they are related in a happens-before relationship A has to finish before B can start. Data races 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 may lead to data races. The outcome of a interrupt-driven program with data races is non-deterministic [5]. The order of execution of operations vary, which may result in bugs that are not reproducible or difficult to reproduce.

This is a simple example of a possible data races between a `main` function and an `isr` function.

```
1   # Data Race Example
2   long shared1 = 0
3
4   def main():
5
6   unsigned tmp = 0
7
8   tmp = shared1
9
10  def isr1():
11
12  idlerun()
13  shared1 = 1
14  idlerun()
15
16  main()
```

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

Static data race detection involves analyzing the source code of a program without executing it to identify potential race conditions, which are situations where the outcome of a program depends on the timing of uncontrollable events like thread execution order.
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 [1].

Dynamic Data Race Detection

Dynamic data race detection, on the other hand, involves monitoring the execution of a program in real-time to detect actual race conditions as they occur, relying on runtime information to identify conflicts in memory access by concurrent threads.
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 [4].

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 detect data races most reliably. However, this work, is 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 [7]. Since this work is 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.

The following code shows an example for a disable and enable call that lead to the safe access of shared data.

```
1   # Data Race Example with ISR Enable/Disable
2   long shared1 = 0;
3
4   def main():
5
6   unsigned tmp = 0;
7   disable_isr(1);
8   tmp = shared1;
9   enable_isr(1);
10  def isr1():
11
12  idlerun();
13  shared1 = 1;
14  idlerun();
15
16  main();
```

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 analyses, especially those who utalize RICFGs, are a promising approach to identify 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. Traversion of the RICFG using a DFS 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, which improves the reliability and correctness of interrupt-driven systems [1].

The methodology for static data race detection in interrupt-driven systems by Wang et al. involves the following key steps.

1. 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.

2. Next, the RICFGs are analyzed to find potential data races, focusing on paths where concurrent access to shared data is done without proper synchronization.

3. Finally, the developer can use the analysis results to address identified data races early in the development process [1].

Algorithm 2: 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 ∈ G_i do
3         for each sv_j ∈ 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 ∪ {< sv_i, sv_j >};
```

The *Static Race Detection* algorithm by Wang et al., presented in Algorithm 2 , takes the RICFGs of program P as input and outputs the potential racing pairs (PR). For each pair of graphs $< G_i; G_j >$ in the RICFGs, the algorithm iterates over each shared variable sv_i in G_i and each shared variable sv_j in G_j. If the variables sv_i and sv_j are the same ($sv_i.V == sv_j.V$), at least one of the accesses is a write operation (($sv_i.A == W$ or $sv_j.A == W$)), the priority of G_i is less than that of G_j ($G_i.pri < G_j.pri$), and the interrupt status table ($INTB$) indicates that the interrupt for sv_i is enabled while sv_j is being accessed, then the pair $< sv_i, sv_j >$ is added to the set of potential racing pairs (PR) [1].

The following introduces the implementation of new a static analysis program based on the static race detection approach of Wang et al.

3 Implementation

In the following, this chapter provides an in-depth explanation of the implementation. For the generation of the input, GNU Compiler Collection (GCC) is used. The command `gcc -fdump-tree-cfg` provides a `cfg-file` with all the important information for the intended data race analysis.

The `cfg-files` are structured as follows. At the start of each function is a commented line with the function name. This line is used to strip the function name in the implementation.

```
;; Function int main()
```

Following the function name, there is summary of every basic block including their successors. This part is used to add the successors of each basic blocks to their initiated items.

```
;;  nodes: 0 1 2 3 4 5 6
;; 2 succs { 3 4 }
;; 3 succs { 5 }
;; 4 succs { 5 }
;; 5 succs { 6 }
;; 6 succs { 1 }
```

Finally there is the actual execution of each function. It displays each basic block and what they do. This part is used to find critical lines that access the shared resource or changing the status of an ISR.

```
int main() ()
{
    int variable2;
    unsigned char tmp;
    int D.1934;
    long int shared1.2;
    long int shared1.1;
    bool retval.0;

    <bb 2>:
    disable_isr (1);
    shared1 = 0;
    shared1.1 = shared1;
    retval.0 = shared1.1 != 0;
    if (retval.0 != 0)
    goto <bb 3>;
    else
    goto <bb 4>;

    <bb 3>:
    enable_isr (1);
    goto <bb 5>;

    <bb 4>:
```

```
25      variable2 = 1;
26
27      <bb 5>:
28      shared1.2 = shared1;
29      tmp = (unsigned char) shared1.2;
30      enable_isr (1);
31      D.1934 = 0;
32
33      <L3>:
34      return D.1934;
35
36  }
```

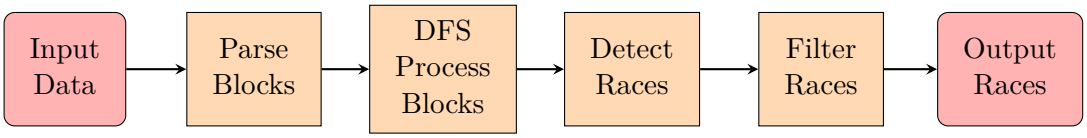

Figure 3.1: Flow Chart of the Program

Figure 3.1 shows the flow of an analysis of such a file. First the input is parsed and generates the basic blocks items, which are then traversed using a DFS and processed while traversing through them. With the informations generated by the traversion of the RICFG possible data races are detected and saved. Finally correct shared data accesses that meet the criterias used in the data race detection are filtered out of the list of data races. The explanation of the implementation is split into the initialization of the basic block class, the parsing of the input, the depth-first search to explore all path, the actual data race analysis, and the filtering of false positives found in the data race analysis.

3.1 Class BasicBlock

The class `BasicBlock` displays all the information necessary for the data race analysis. The basic blocks build the RICFG by storing every possible path of the functions in its successors. Each BB item also stores all the important informations for the further data race analysis like priority of the function, shared resource access in the BB, enable or disable calls of ISRs and function calls for inter-procedural edges between the function. Those information include the following attributes:

- `function_name`: The function name to which the basic block belongs.
- `number`: The number of the basic block.
- `priority`: The priority of the function the BB is in.
- `shared_resources`: All accesses of shared resources within the 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 diagram in Figure 3.2:

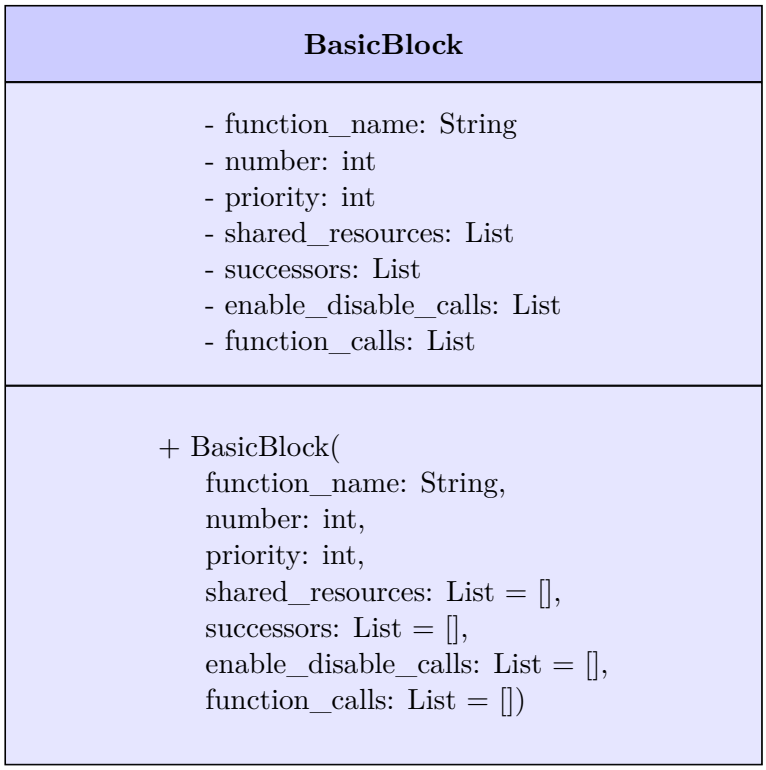

Figure 3.2: UML: Class BasicBlock

3.2 Parsing and Helper Functions

This section explains the parsing of the input and the helper functions, which are called in the data race analysis.

The parsing of the input iterates two times through the code lines to extract all the important information of the code and save it in the BB items. The first iteration of the code lines initiates the BBs with the BB number and the function it relates to. Using these regular expression:

```
1     func_match = re.match(r';; Function (.+?) \(', line)
2     bb_match = re.match(r'<bb (\d+)>:', line)
```

Additionally, it adds the information of shared resources, enable/disable calls of ISRs and function calls within the BB. To identify the shared reources and their operation type these regular expression are used:

```
1    re.search(fr'\b{resource_name}\b', line)
2    re.search(fr'\b{resource_name}\b\s*=', line) and not re.search(fr'\b{
         resource_name}\b\s*==', line)
```

The second iteration adds the successors of the BB, using this regular expression:

```
1    succ_match = re.match(r';; (\d+) succs \{(.+?)\}', line)
```

A second iteration is used to ensure all the BBs items are initialized first to ensure a correct handling of the successors.

There are also multiple helper functions, which are called during the data race analysis part of the implementation.

The `determine_priority` function is determining the priority of the function which is involved in a possible data race. Since one condition for a data race is that the interrupting function has a higher priority, this is an important check. The function uses a regular expression to find the number of a ISR to use that as its priority.

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

The priority is determined in the first place by differentiating between ISR functions and normal functions because ISRs always have a higher priority than non-ISR functions. In this implementation non-ISR functions have the priority infinity and ISRs are ordered by the number of it, while lower number ISRs have a higher priority than higher number ones.

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.

3.3 Data Race Detection

The following algorithms 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. The algorithm are ordered as they are found in the code. All the following algorithms are part of the `detect_data_race` function in the code. The DFS algorithm is exploring all possible paths thorugh the CFG while calling the Algorithm 4 Process Block, which processes the shared resources and enable or disable calls within the BB in order. It appends the shared resources to a list `resource_access` and updates the `current_isr_status`. The resource access list is then used in the Alogrithm 7 Check for Data Races, where it finds possible data races using the critical criterias of a data race. In the end the Algorithm 8 Filter Data Races is using the `isr_status` aswell as the `isr_enabling_map` to filter out correctly accessed shared resources.

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.

Algorithm 3: ISR Enabling Map

Data: blocks
Result: List ISR enabling ISRs filled
Input: Dictionary of BasicBlocks
Output: List of enabling ISRs

```
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 isr number 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 main loop of the Algorithm 3 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 4: 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  combined_events ← sort(block.shared_resources + block.enable_disable_calls,
     key=lambda x: x[2] if len(x) == 3 else x[1])
2  foreach event in combined_events do
3      if event is a shared resource access (len(event) == 3) then
4          resource_name, access_type, res_line_number ← event
5          resource_accesses[resource_name].append((block.function_name,
             block.number, access_type, res_line_number,
             current_isr_status.copy(), block.priority))
6      end
7      else if event is an enable/disable ISR call (len(event) == 2) then
8          call, line_number ← event
9          isr_idx_match ← search for ISR number in call
10         if isr_idx_match is found then
11             isr_idx ← integer value of the first group in isr_idx_match minus one
12             if "disable_isr" is in call then
13                 if 0 ≤ isr_idx < length of current_isr_status then
14                     current_isr_status[isr_idx] ← 1
15                 end
16             else
17                 if 0 ≤ isr_idx < length of current_isr_status then
18                     current_isr_status[isr_idx] ← 0
19                 end
20             end
21         end
22     end
23 end
```

Algorithm 4 iterates through each line of a basic block to find the current ISR status while resources are accessed. It sorts each access of shared resources and the enable or disable calls of ISRs by line number to ensure the correct order of execution. The algorithm differs between shared resource access and ISR enable or disable calls by the length of the event. Accesses have three entries, the resource name, the access type and the line number and the enable and disable calls have two entries the call and the line number. 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. If an enable or disable call is found the algorithm changes the corresponding bit in the `current_isr_status` array to a 1 for diable calls and to a 0 for enable calls. All this information is used later for the detection and filter of data races.

Algorithm 5: DFS and Initialization

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

```
1  Function dfs(block, visited_blocks, current_isr_status, path):
2      if (block.function_name, block.number) in visited_blocks then
3          block_isr_statuses[(block.function_name, block.number)] ←
             merge_isr_statuses(block_isr_statuses[(block.function_name,
             block.number)], current_isr_status)
4          return
5      end
6      visited_blocks.add((block.function_name, block.number))
7      path.append((block.function_name, block.number))
8      block_isr_statuses[(block.function_name, block.number)] ←
         current_isr_status.copy()
9      process_block(block, current_isr_status)
10     if not block.successors then
11         return
12     end
13     else
14         for successor in block.successors do
15             dfs(successor, set(visited_blocks), current_isr_status.copy(),
                 path.copy())
16         end
17     end
   // Initialization and starting DFS from basic blocks with number 2
18 for (func_name, bb_num), block in blocks.items() do
19     if bb_num == 2 then
20         initial_isr_status ← track_isr_status(blocks).copy()
21         process_block(block, initial_isr_status)
22         for successor in block.successors do
23             dfs(successor, set(), initial_isr_status.copy(), [(func_name,
                 bb_num)])
24         end
25     end
26 end
```

The Algorithm 5 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. The DFS always is initiated at BB number two.

Algorithm 6: 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

```
1 merged_status ← empty list
2 for isr1, isr2 in zip(isr_status1, isr_status2) do
3 |   merged_status.append(min(isr1, isr2))
4 end
5 return merged_status
```

Alogrithm 6 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 7: Check for Data Races

Data: resource_accesses
Result: List of potential data races
Input: Dictionary of resource accesses
Output: List of potential data races

```
1  Function check_for_data_races():
2      for resource, accesses in resource_accesses.items() do
3          for i, (func1, bb_num1, access_type1, line_number1, isr_status1,
             priority1) in enumerate(accesses) do
4              for j, (func2, bb_num2, access_type2, line_number2, isr_status2,
                 priority2) in enumerate(accesses) do
5                  if i ≥ j then
6                      Continue
7                  end
8                  if func1 ≠ func2 and (access_type1 == "write" or access_type2
                     == "write") and priority1 ≠ priority2 then
9                      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)))
10                 end
11             end
12         end
13     end
```

The Algorithm 7 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.

3.4 Filter of Possible Data Races

Algorithm 8: 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

```
1  filtered_data_races ← empty list
2  seen_races ← empty set
3  for resource, access1, access2 in potential_data_races do
4      func1, line_number1, isr_status1, ← access1
5      func2, line_number2, isr_status2, ← access2
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      race_key ← frozenset(((func1, line_number1), (func2, line_number2)))
9      if not (relevant_isr_disabled1 or relevant_isr_disabled2) and race_key not
         in seen_races then
10         filtered_data_races.append((resource, access1, access2))
11         seen_races.add(race_key)
12     end
13 end
14 return filtered_data_races
```

The Algorithm 8 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 9: 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

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

The Algorithm 9 checks if the bit corresponding to the ISR in the ISR status array is set to one. An ISR is disabled with a one in its corresponding bit in the ISR status list and enabled with an 0. If so, the function returns true to the `filter_data_races` function, and if not, it returns false.

Algorithm 10: 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

```
1  isr_idx ← extract_isr_index(func_name)
2  if isr_idx is not None then
3      for enabler_isr, enabled_isrs in isr_enabling_map.items() do
4          enabler_idx ← extract_isr_index(enabler_isr)
5          if enabler_idx is not None and not is_isr_disabled(isr_status,
             enabler_isr) then
6              if isr_idx in enabled_isrs then
7                  return True
8              end
9          end
10     end
11 end
12 return False
```

The Algorithm 10 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.

4 Evaluation

The efficiency and effectiveness of the implemented static data race detection framework were evaluated using the benchmarks provided by the racebench 2.1 GitHub repository[1]. To cover a wider range of scenarios, some benchmarks were added to specifically evaluate considered cases. The benchmarks are all manually checked for data races to compare the expected data races found and the actual number found by the analysis tool. Since part of the work is also the reduction of the computing overhead by reducing the ICFGs, the analysis time of the program is also added to the evaluation.

Benchmark	**#ISR**	**#Func**	**#SR**	**#BB**
Simple Enable/Disable Calls	2	3	1	7
Multiple Resources	2	3	2	7
ISR Enabling	2	3	1	7
Function Calls	2	4	1	5
ISR Enabling 2	3	4	4	13
ISR Enabling Depth	3	4	4	13
svp_simple_006_001	1	2	2	24
svp_simple_012_001	1	2	1	2
svp_simple_014_001	3	4	4	13
svp_simple_019_001	1	2	8	15

Table 4.1: Objects of Analysis

Table 4.1 shows a summary of the important characteristics of the used benchmarks. The number of ISRs, functions, shared resources, and basic blocks is shown for each CFG. These numbers help to have a brief understanding of the depth of each function without actually understanding the `cfg-file` of each of those benchmarks.

Benchmark	**Manual**	**Program**	**Time (seconds)**
Simple Enable/Disable Calls	2	2	0.007
Multiple Resources	3	3	0.004
ISR Enabling	2	2	0.006
Function Calls	4	4	0.005
ISR Enabling 2	8	8	0.004
ISR Enabling with Depth	8	4	0.009
svp_simple_006_001	4	4	0.002
svp_simple_012_001	2	1	0.002
svp_simple_014_001	8	8	0.004
svp_simple_019_001	10	10	0.006

Table 4.2: Comparison of Manual and Program-Detected Data Races

Table 4.2 displays the results of the evaluation with suited CFGs . The benchmarks are chosen to show most of the cases that were considered when developing the data race detection framework. Those evaluated cases include simple enable/disable calls, access

[1] https://github.com/chenruibuaa/racebench/tree/master/2.1

of multiple shared resources, ISR enabling ISRs, inter-procedural function calls and deep if-else chains. The execution times are determined with the `time` module in python and it tracks the time after entering the input until completed execution of the analysis. The program reliably detects data races within the considered scenarios. It successfully keeps track of every enable and disable call of ISRs as well as considering all the other criteria for a data race shown earlier like priority, access type, and so on. In the program `ISR Enabling with Depth`, an ISR enabling map with a depth of two would be needed to detect all eight data races, which is not included in this work. `svp_simple_012_001` uses pointers, which also leads to false negatives with the current implementation. All the used benchmarks can be found in the GitHub reposetory of this work[2]. The runtime of the program stays consistently low within the testing with the benchmarks of racebench 2.1.

[2]https://github.com/RobinWillenbrock/BADataRaces/tree/main/Code/Benchmarks

5 Discussion

This thesis presents a static analysis framework for detecting data races in interrupt-driven software using reduced inter-procedural control flow graphs. The approach efficiently identifies potential data races by analyzing the software's source code and focusing on essential control flow elements and code lines. This discussion highlights the completed aspects of the implementation, the current limitations, and the potential enhancements of the analysis program.

The framework detects data races effectively by meeting the essential criteria, such as identifying shared resources in the input and their operation type as well as the priority they are used with.

The provided `cfg-file` is interpreted and used to develop a reduced inter-procedural control flow graph. The interprocedural calls of functions are stored in the basic block items and used to analyze the edges between functions. In the current implementation, the critical information of called functions is added to the currently processed BB. This can be further improved by adding the calls into the DFS to ensure all possible interactions are considered. This could improve the stability of the inter-procedural analysis compared to the current implementation.

The reduction of the generated ICFG is realized by reducing the information each basic block carries to a minimum needed for the analysis. Only the critical lines of the CFG get stored in the basic blocks, which form the RICFG. To ensure a flawless path through the graph, the nodes are not deleted when they do not carry any important information but are left empty to minimize the computational overhead the program has to execute. Nonetheless, it can be further optimized by completely removing the nodes and passing the successors. For larger CFGs, this would be an improvement, but for smaller CFGs, the current approach is comparable because it does not add the computation of the successor handling.

The CFGs are traversed with a DFS algorithm that ensures all possible paths are discovered. In a DFS, one missing successor can lead to an incomplete path and result in missing possible data races. As mentioned earlier, this is the reason this work focuses on the correctness of the path over the possible reduction of the computation by not deleting the BB completely and keeping them empty with only the successors.

Additionally, the ISR status array is implemented and dynamically updated through the CFG traversal. By adding the enable and disable calls of a basic block with their corresponding code lines to the item, an efficient way of traversing through the CFG is enabled while keeping the execution flow correct even within a basic block. The array ensures a correct handling of every ISR status change. This results in a significant reduction of false positives and an easier interpretation of the data races that are found.

As an improvement to the prior work of Wang et al., this work implemented a solution to ISR enabling ISRs. By introducing an ISR enabling map, all the ISR enable operations of other ISRs are considered. This leads to detected data races with the corresponding ISR being disabled, which can then be further analyzed by the developer.

All in all, the program stores the minimal information needed for the analysis in

different basic block items which form an RICFG. This RICFGs is then traversed by a DFS algorithm to find all possible data races while considering the ISR status at any point, including ISR enabling ISRs.

The current implementation does not include a pointer analysis, which leads to possible false negatives. The framework can be further improved by adding a pointer analysis, which would improve the precision of the tool. Since there are several tools for such analysis, this work focused on the implementation of other critical parts of the data race analysis such as the ISR handling.

Furthermore, the handling of ISR enabling ISRs is currently limited to a depth of one. The program can be further improved to find possible enable operations of other ISRs in greater depth. Since the scenarios of multiple ISRs enabling each other consecutively in the correct order to lead to a data race are very rare, this work is not considering those cases.

6 Conclusion

This thesis presents a static analysis tool for detecting data races in interrupt-driven software. By using reduced inter-procedural control flow graphs, the tool effecitvely identifies potential data races by focusing on essential control flow elements and shared resource accesses. The DFS alogrithm ensures exploration of all possible paths, while keeping track of the ISR status of each ISR at any point of the control flow.

The evaluation of the framework using a variety of benchmarks, including the ones from the racebench 2.1 repository and self optimized ones to display special cases, demonstrate the effiency and accuracy of the program in different scenarios. The results show a reliable identification of data races while maintaining a low computional overhead.

The current implementation still has limitations, such as the handling of pointers and the depth of ISR enabling scenarios. Future work could focus on then integration of a pointer analysis tool to further improve the precision of the framework. On top of that the ISR enabling map can be improved by considering the enable operations in further depth. Additionally the implementation of the RICFG can be further optimized by completely removing basic blocks without important informations while maintaining the correct control flow.

In conclusion, the developed static data race detection framework provides a valuable tool, which enables early detection of data races in interrupt-driven software.

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

[8] 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)
```

```
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)
192     relevant_isr_disabled2 = is_isr_disabled(isr_status2, func1) and not
            is_isr_enabled_by_another(isr_status2, func1)
193
194     if not (relevant_isr_disabled1 or relevant_isr_disabled2):
```

```
195     filtered_data_races.append((resource, access1, access2))
196
197     return filtered_data_races
198
199     filtered_data_races = filter_data_races(potential_data_races)
200
201     return filtered_data_races
202
203
204     shared_resource_input = input("Enter the names of shared resources,
            separated by commas: ")
205     shared_resource_names = [name.strip() for name in
            shared_resource_input.split(',')]
206
207     file_path = input("Enter the file path: ").strip()
208     blocks = parse_basic_blocks(file_path, shared_resource_names)
209
210     data_races = detect_data_races(blocks)
211
212     print("Detected Data Races:")
213     for resource, access1, access2 in data_races:
214     print(f"Resource: {resource}")
215     print(f"  Access 1: Function {access1[0]} (BB {access1[1]}), {access1
            [2]}, Line {access1[3]}, ISR Status: {access1[4]}")
216     print(f"  Access 2: Function {access2[0]} (BB {access2[1]}), {access2
            [2]}, Line {access2[3]}, ISR Status: {access2[4]}")
217     print()
```