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 9, 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

Declaration of Originality[1]

Hereby I confirm, Robin Willenbrock, Matr. Nr. 53962, that this assignment is my own work and that I have only sought and used mentioned tools.
I have clearly referenced in the text and the bibliography all sources used in the work (printed sources, internet or any other source), including verbatim citations or paraphrases.
I am aware of the fact that plagiarism is an attempt to deceit which, in case of recurrence, can result in a loss of test authorization.
Furthermore, I confirm that neither this work nor parts of it have been previously, or concurrently, used as an exam work – neither for other courses nor within other exam processes.

Hamburg, July 10, 2024

Robin Willenbrock

[1] University of Tübingen, Department of Sociology

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. It is common for such systems to utilise interrupt service routines (ISRs) in order to achieve 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.

The prevention of these issues is of vital importance for the reliable and correct execution of software, and thus the detection of data races is of great significance. 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 analysing the graphs, it is possible to identify paths where shared resources are accessed concurrently without proper synchronisation, which may indicate the presence of data races.The paths are explored by using a depth-first search (DFS) algorithm, which ensures the traversion of every possible node in every path. Additionally, a mechanism to analyze enabling and disabling ISRs, a common practice to ensure data consistency in interrupt-driven systems, is implemented.

Structur

This thesis provides a chapter with important backround information to help understand the overall problem of data races. The backround chapter is giving an 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 program.

The implementation is then evaluated by using public benchmarks and self generated ones to cover a wide spectrum of possible cases. On top of that the time complexity of the program is analyzed.

Following the evaluation the results of the program are discussed, including an overview of the program and showing possible improvements to the analysis framework that could be implemented in future work.

In the end the thesis is finalized in a 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 inter-procedural control flow graphs. This information includes basics about interrupt-driven systems, shared resources, RICFGs, data races as a whole and GNU Compiler Collection (GCC).

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].

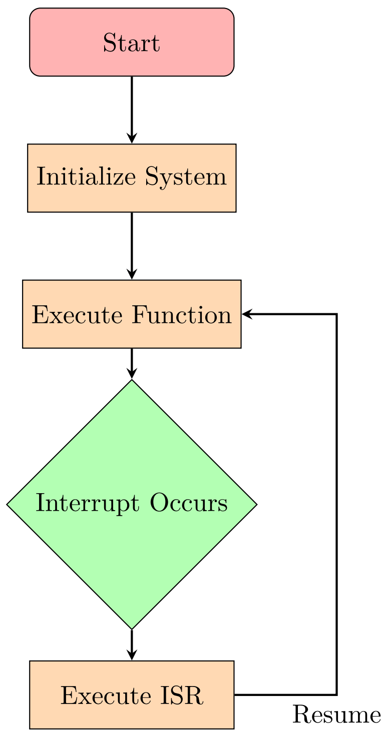

Figure 2.1: Flowchart of the Interrupt-Driven System

In Figure 2.1, a basic execution flow of a 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 represents the most challenging aspect of an interrupt-driven system, as it is crucial to maintain the system's fast responsiveness. 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 an interrupt-driven system. One of the biggest challenges is the correct handling of the different priorities of the interrupts.[1] To conclude in an interrupt-driven system, the execution of the main program and ISRs 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 [6].

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 [6]. 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 (CFG) 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 [7].
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, which represent the control flow within a single function.
  - Call edges, which represent the calling of a function [7].

An RICFG is an optimized version of the ICFG that simplifies the graph to only the necessary information needed for the analysis [1].

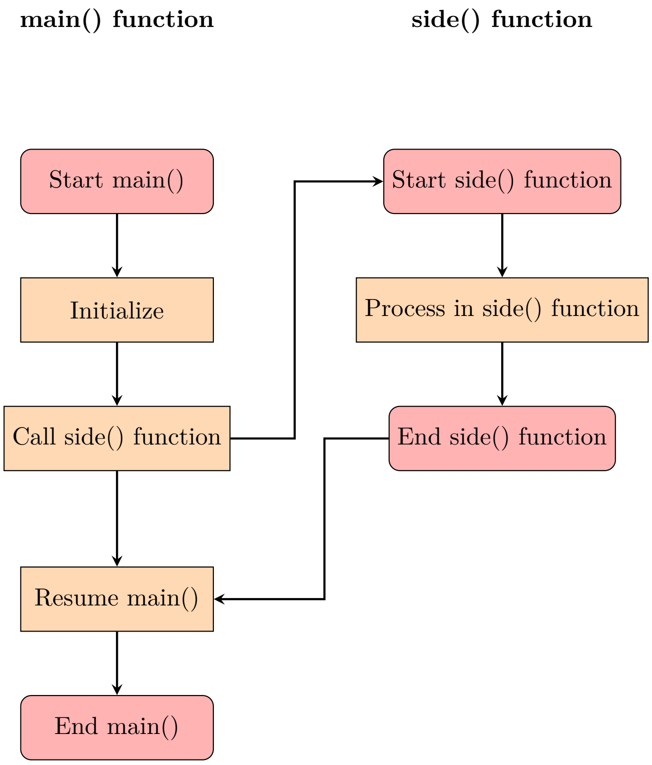

Figure 2.2: Example of an Inter-Procedural Control Flow Graph

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, 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 maintaining sufficient detail, RICFGs are a effective tool for the static analysis of data races [5][1].

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, the reduction enhances the scalability of static analysis and make it more practical to analyze more complex data structures [1].

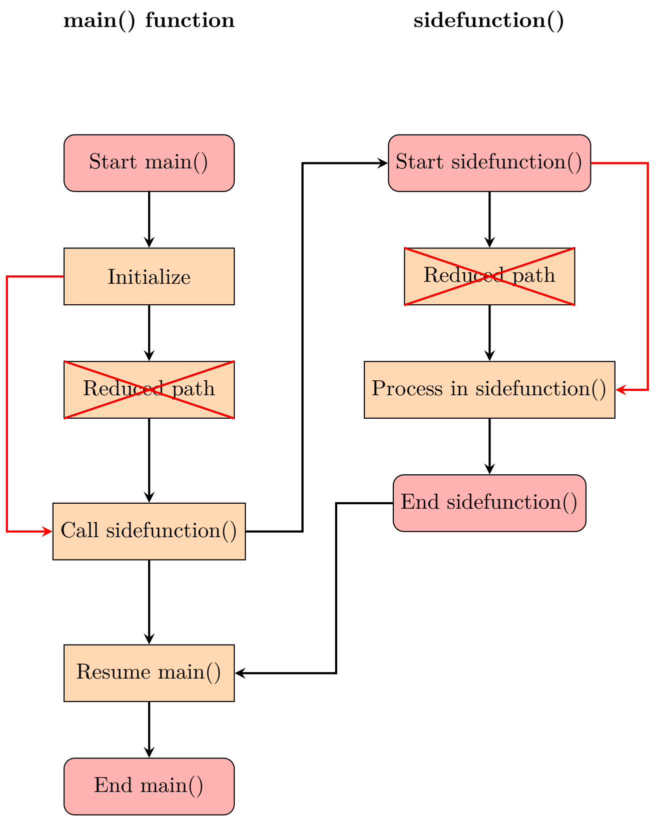

Figure 2.3: Example of a Reduced Inter-Procedural Control Flow Graph

Figure 2.3 shows the example of a simple ICFG with added unneccessary nodes. The resulting RICFG reduces the control flow graph (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

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

Algorithm 1 shows the algorithm that is used to execute an DFS. 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 [7].

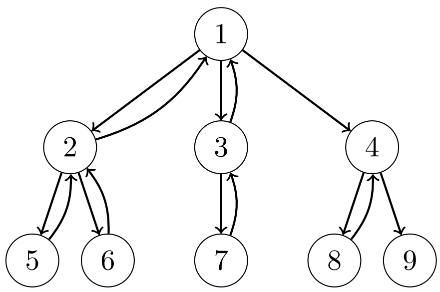

Figure 2.4: Depth-First Search of a Tree

Figure 2.4 displays the execution of a DFS. Starting from node 1 the algorithm goes to the next node here node 2. From there it traverses to the next node in line, which is linked with node 2. Node 5 is the end of the branch and has no successors. From a node like this the algorithm backtracks to the last node that had a successor, here shown with a curved arrow. This process repeats through the whole tree resulting in the traversion of the tree in the following order: 1,2,5,6,3,7,4,8,9.

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 [4]. 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. The order of execution of operations vary, which may result in bugs that are not reproducible or difficult to reproduce [4].

```
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()
```

Figure 2.5: Simple Data Race Example

Figure 2.5 shows a simple example of a possible data race between a `main` function and an `isr` function. 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 [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 [1].

- 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 [4].

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 [4].
- 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 [2].
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 [2].
- Context-Sensitive Detection: By analyzing the actual runtime behavior, dynamic analysis can understand the context of operations, leading to more accurate detection [2].

Disadvantages:

- Performance Overhead: The analysis at runtime can slow down the application significantly [4].
- 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 [2].

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.[2] However, this work, focuses 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 [6]. Since this work focuses 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 correct analyzation of those synchronization mechanisms prevents false posititves.

```
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();
```

Figure 2.6: Data Race Example with ISR Enable/Disable

Figure 2.6 shows an example for a disable and enable call that lead to the safe access of 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 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. [1] 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)

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**
        $PR = PR \cup \{< sv_i, sv_j >\}$;

The *Static Race Detection* algorithm by Wang et al. [1], 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 chapter introduces the implementation of a new static analysis program based on the static race detection approach of Wang et al. [1].

2.6 GNU Compiler Collection

For the generation of the input, GCC[1] is used. GCC is a suite for compilers for various programming languages, including C and C++, among others. The command `gcc -fdump-tree-cfg` generates a `cfg-file` out of a program file, with all the important information for the intended data race analysis.

[1]https://gcc.gnu.org/

```
1  ;; Function int main()
2
3  ;;  nodes: 0 1 2 3 4 5 6
4  ;; 2 succs { 3 4 }
5  ;; 3 succs { 5 }
6  ;; 4 succs { 5 }
7  ;; 5 succs { 6 }
8  ;; 6 succs { 1 }
9
10 int main() ()
11 {
12     int variable2;
13     unsigned char tmp;
14     int D.1934;
15     long int shared1.2;
16     long int shared1.1;
17     bool retval.0;
18
19     <bb 2>:
20     disable_isr (1);
21     shared1 = 0;
22     shared1.1 = shared1;
23     retval.0 = shared1.1 != 0;
24     if (retval.0 != 0)
25     goto <bb 3>;
26     else
27     goto <bb 4>;
28
29     <bb 3>:
30     enable_isr (1);
31     goto <bb 5>;
32
33     <bb 4>:
34     variable2 = 1;
35
36     <bb 5>:
37     shared1.2 = shared1;
38     tmp = (unsigned char) shared1.2;
39     enable_isr (1);
40     D.1934 = 0;
41
42     <L3>:
43     return D.1934;
44 }
```

Figure 2.7: Example CFG generated with GCC

Figure 2.7 shows an reduced example of a `cfg-file` generated with GCC, which displays the later used parts of the `cfg-file`. In line 1 of each `cfg-file` is a commented line with the function name. This line is used to strip the function name in the implementation.

Following the function name, in line 3 to 8, 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.

Finally there is the actual execution of each function in line 10 to 44. 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.

3 Implementation

The following chapter provides an in-depth explanation of the implementation.

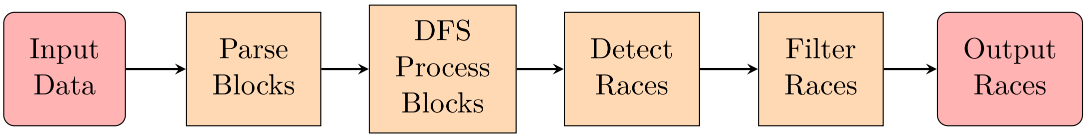

Figure 3.1: Flow Chart of the Program

Figure 3.1 shows the flow of the data race analysis. 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 information 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.

To summarize, Figure 3.2 shows the UML diagram of the `BasicBlock` class.

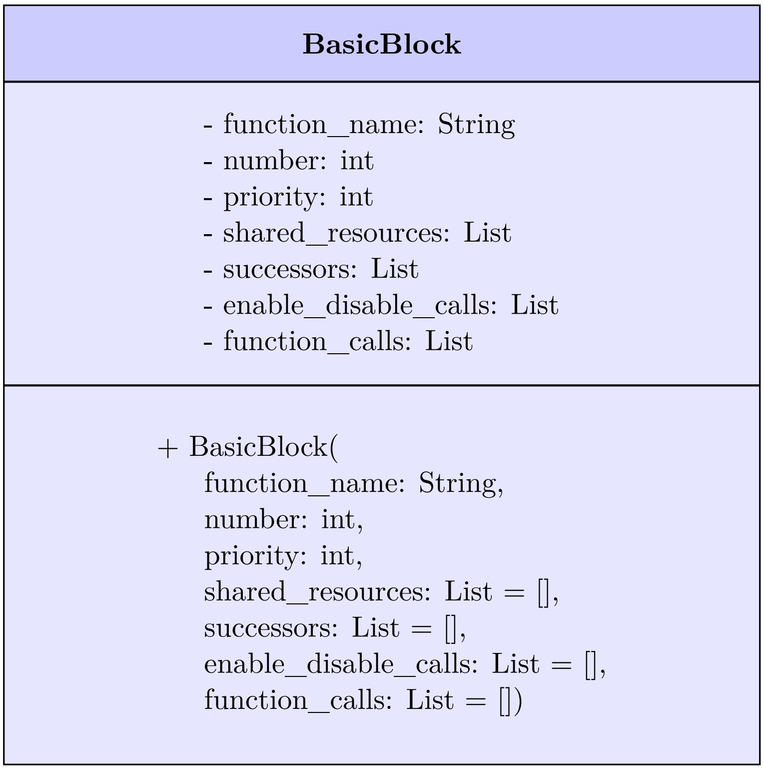

Figure 3.2: UML: Class BasicBlock

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:

```
func_match = re.match(r';; Function (.+?) \(', line)
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:

```
re.search(fr'\b{resource_name}\b', line)
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:

```
    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.

```
    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 algorithms are ordered as they are found in the code. All the following algorithms are part of the `detect_data_race` function in the program. 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 `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

```
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 isr number 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
```

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 of 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

combined_events ← sort(block.shared_resources + block.enable_disable_calls,
  key=lambda x: x[2] if len(x) == 3 else x[1])
foreach event in combined_events do
    if event is a shared resource access (len(event) == 3) then
        resource_name, access_type, res_line_number ← event
        resource_accesses[resource_name].append((block.function_name,
          block.number, access_type, res_line_number,
          current_isr_status.copy(), block.priority))
    end
    else if event is an enable/disable ISR call (len(event) == 2) then
        call, line_number ← event
        isr_idx_match ← search for ISR number in call
        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 call 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
end
```

Algorithm 4 iterates through each shared resource access and enable/disable call 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. 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 1 for diable calls and to 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

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
```

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

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

The Algorithm 5 is recursively traversing each block in a possible path of the RICFG, while calling the `process_block` function. 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, shown in Algorithm 6. It 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

```
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
```

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

```
filtered_data_races ← empty list
seen_races ← empty set
for resource, access1, access2 in potential_data_races do
    func1, line_number1, isr_status1, ← access1
    func2, line_number2, isr_status2, ← 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
```

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. The algorithm also deletes possible duplicates of a data race. The output is the final list of detected data races.

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

```
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
```

The Algorithm 9 checks if the bit corresponding to the ISR in the ISR status array is set to 1. An ISR is disabled with a 1 in its corresponding bit in the ISR status list and enabled with an 0.

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

```
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
```

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.

Summarize Implementation

A full execution of all the algorithms takes a `cfg-file` as input and parses it to extract all the important information. Those information are saved in BB items, which are used to build a RICFG. The resulting RICFG is traversed using a DFS algorithm. The BB in each path get processed and the critical sections of the path are analyzed. The found accesses to shared resources are then analyzed for possible data races. Those found data races are then filtered and the resulting races are saved in a list. This list is printed to output the possible data races including the functions, BBs and the line numbers of the accesses.

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 are added to specifically evaluate considered cases. The benchmarks are all manually checked for data races to compare the expected amount of data races and the actual number found by the analysis program. 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. To ensure there are no side effects in the system impacting the time output each benchmark is run multiple times to find the consistent execution time.

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. Since no pointer analysis is done. 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. All of the evaluation has been performed on a PC with 6-core Intel Core CPU i5-12400F (2.5GHz) and 16GB RAM on Windows 10.0.19045.

Time Complexity Analysis

Parsing and Setting Up

- `parse_basic_blocks(file_path, shared_resource_names)`:

$$\mathcal{O}(m \cdot k)$$

where m is the number of lines in the file and k is the number of shared resources.

- `propagate_function_calls(blocks, function_blocks)`:

$$\mathcal{O}(f \cdot b \cdot c)$$

where f is the number of function blocks, b is the number of basic blocks, and c is the number of calls per block.

Detecting Data Races

- `detect_data_races(blocks)`:

$$\mathcal{O}(b \cdot (e + r + s))$$

where b is the number of blocks, e is the number of enable/disable calls, r is the number of shared resource accesses, and s is the number of successors per block.

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

- `check_for_data_races()`:
  $$\mathcal{O}(n \cdot a^2)$$
  where n is the number of resources and a is the number of accesses to a resource.
- `filter_data_races(potential_data_races)`:
  $$\mathcal{O}(d)$$
  where d is the number of potential data races.
- `process_block(block, current_isr_status)`:
  $$\mathcal{O}(e \log e)$$
- `dfs(block, visited_blocks, current_isr_status, path)`:
  $$\mathcal{O}(b + s)$$

Combined Complexity

With consideration of the relationships between the variables, the combined overall time complexity is:

$$\mathcal{O}(m \cdot k + b \cdot (f \cdot c + e + r + s + e \log e) + n \cdot a^2)$$

The most critical variables are b and a. The value of b is multiplied by a number of factors, resulting in a significant increase in the scaling if b is large. The term a is of particular importance as it is the only quadratic scaling term in the complexity analysis. In certain conditions, the other terms may become dominant in determining the time complexity.

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 all accesses of the provided shared resources 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 program adds the critical information of called functions to the currently processed BB and simulates a traversion of the called function that way. 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. They only hold their successors to minimize the computational overhead the program has to execute. 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. 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.

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.

Future Work and Improvements

The following section shows possible improvements for the program that did not fit in the current work.

The global variables have to be known and provided as input in the current implementation. Adding a way to automatically find the shared resources could further improve the program.

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 focuses on the implementation of other critical parts of the data race analysis such as the ISR handling.

The RICFG can be further optimized by completely removing the nodes and passing the successors to the prior BB. 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.

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.

For a more consistent data race analysis an addition of a dynamic analysis tool to complement the static analysis would be optimal. As already shown in the Chapter 2.4.1 both approaches have advantages and disadvantages but implementing both, yields the best results.

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 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 by minimizing the analyzed data.

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.

The objective of this thesis was the development of a program for static data race analysis in interrupt driven software using reduced inter-procedural control flow graphs. In conclusion, the developed static data race detection framework provides a valuable framework, 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] R. Chen, Xiangying Guo, Y. Duan, B. Gu, Mengfei Yang (2011). Static Data Race Detection for Interrupt-Driven Embedded Software.

[5] Gold, R. (2014). Reductions of Control Flow Graphs.

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

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

Attachments

```
import re
from collections import defaultdict
import time

# Input from user
shared_resource_input = input("Enter the names of shared resources, separated by commas: ")
file_path = input("Enter the file path: ").strip()

start_time = time.time()

# Definition of BasicBlock class
class BasicBlock:
def __init__(self, function_name, number, priority, shared_resources=None, successors=None, enable_disable_calls=None, function_calls=None):
self.function_name = function_name
self.number = number
self.priority = priority
self.shared_resources = shared_resources if shared_resources else []
self.successors = successors if successors else []
self.enable_disable_calls = enable_disable_calls if enable_disable_calls else []
self.function_calls = function_calls if function_calls else []

def __repr__(self):
return (f"BasicBlock(function_name={self.function_name}, number={self.number}, priority={self.priority}, shared_resources={self.shared_resources}, "
f"successors={[succ.number for succ in self.successors]}, enable_disable_calls={self.enable_disable_calls}, function_calls={self.function_calls})")

# Parse BasicBlocks from input file
def parse_basic_blocks(file_path, shared_resource_names):
blocks = {}
function_blocks = defaultdict(list)
current_function = None

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

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

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

```

```
#Functions
func_match = re.match(r';; Function (.+?) \(', line)
if func_match:
if bb_num is not None and current_function is not None:
priority = determine_priority(current_function)
blocks[(current_function, bb_num)] = BasicBlock(current_function, bb_num, priority, shared_resources, [], enable_disable_calls, function_calls)
function_blocks[current_function].append(blocks[(current_function, bb_num)])
current_function = func_match.group(1)
bb_num = None
continue

# BasicBlocks
bb_match = re.match(r'<bb (\d+)>:', line)
if bb_match:
if bb_num is not None and current_function is not None:
priority = determine_priority(current_function)
blocks[(current_function, bb_num)] = BasicBlock(current_function, bb_num, priority, shared_resources, [], enable_disable_calls, function_calls)
function_blocks[current_function].append(blocks[(current_function, bb_num)])
bb_num = int(bb_match.group(1))
shared_resources = []
enable_disable_calls = []
function_calls = []

# Shared Resources
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) and not 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))

#Enable/Disable calls
if 'enable_isr' in line or 'disable_isr' in line:
enable_disable_calls.append((line.strip(), line_number))

# Function calls
call_match = re.match(r'.*call.*\b(\w+)\b', line)
if call_match:
function_calls.append((call_match.group(1), line_number))

if bb_num is not None and current_function is not None:
priority = determine_priority(current_function)
blocks[(current_function, bb_num)] = BasicBlock(current_function, bb_num, priority, shared_resources, [], enable_disable_calls, function_calls)
function_blocks[current_function].append(blocks[(current_function, bb_num)])

#Second pass to link successors
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, function_blocks

def determine_priority(function_name):
match = re.search(r'isr[_]?(\d+)', function_name)
if match:
return int(match.group(1))  # Higher priority for lower ISR number
return float('inf')  # Lower priority for non-ISR functions

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 merge_isr_statuses(isr_status1, isr_status2):
return [min(isr1, isr2) for isr1, isr2 in zip(isr_status1, isr_status2)]

#Merges shared resources and isr calls into calling block
def propagate_function_calls(blocks, function_blocks):
for func_name, block_list in function_blocks.items():
for block in block_list:
for called_func, line_number in block.function_calls:
if called_func in function_blocks:
for called_block in function_blocks[called_func]:
block.shared_resources.extend(called_block.shared_resources)
block.enable_disable_calls.extend(called_block.enable_disable_calls)

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

```
isr_enabling_map = defaultdict(set)
block_isr_statuses = defaultdict(list)

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

def process_block(block, current_isr_status):
combined_events = sorted(
block.shared_resources + block.enable_disable_calls,
key=lambda x: x[2] if len(x) == 3 else x[1]
)

for event in combined_events:
if isinstance(event, tuple) and len(event) == 3:  # This is a shared resource access
resource_name, access_type, res_line_number = event
resource_accesses[resource_name].append((block.function_name, block.number, access_type, res_line_number, current_isr_status.copy(), block.priority))
elif isinstance(event, tuple) and len(event) == 2:  # This is an enable/disable isr call
call, line_number = event
isr_idx_match = re.search(r'\((\d+)\)', call)
if isr_idx_match:
isr_idx = int(isr_idx_match.group(1)) - 1  # Convert to 0-based index
if "disable_isr" in call:
if 0 <= isr_idx < len(current_isr_status):
current_isr_status[isr_idx] = 1
elif "enable_isr" in call:
if 0 <= isr_idx < len(current_isr_status):
current_isr_status[isr_idx] = 0

def dfs(block, visited_blocks, current_isr_status, path):
if (block.function_name, block.number) in visited_blocks:
block_isr_statuses[(block.function_name, block.number)] = merge_isr_statuses(
block_isr_statuses[(block.function_name, block.number)], current_isr_status)
return
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:
```

```
return
else:
for successor in block.successors:
dfs(successor, set(visited_blocks), current_isr_status.copy(), path.copy())
#Start DFS at bb number 2
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, priority1) in enumerate(accesses):
for j, (func2, bb_num2, access_type2, line_number2, isr_status2, priority2) in enumerate(accesses):
if i >= j:
continue
if func1 != func2 and (access_type1 == "write" or access_type2 == "write") and priority1 != priority2:
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)))

check_for_data_races()

def filter_data_races(potential_data_races):
filtered_data_races = []
seen_races = set()

for resource, access1, access2 in potential_data_races:
func1, bb_num1, access_type1, line_number1, isr_status1, priority1 = access1
func2, bb_num2, access_type2, line_number2, isr_status2, priority2 = 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)

#Filter duplicates
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:
filtered_data_races.append((resource, access1, access2))
seen_races.add(race_key)

return filtered_data_races

filtered_data_races = filter_data_races(potential_data_races)

return filtered_data_races


shared_resource_names = [name.strip() for name in shared_resource_input.
    split(',')]


blocks, function_blocks = parse_basic_blocks(file_path,
    shared_resource_names)


propagate_function_calls(blocks, function_blocks)

data_races = detect_data_races(blocks)

#Ouput
print("\nDetected 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]}, Priority: {access1
    [5]}")
print(f"  Access 2: Function {access2[0]} (BB {access2[1]}), {access2
    [2]}, Line {access2[3]}, ISR Status: {access2[4]}, Priority: {access2
    [5]}")
print()

#Execution Time
end_time = time.time()
execution_time = end_time - start_time
print(f"Execution time: {execution_time} seconds")
```