TUHH
Hamburg
University of
Technology

Bachelor Thesis

Robin Willenbrock

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

June 25, 2024

supervised by:
Ulrike Engeln

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

STS
Software
Technology
Systems

Contents

List of Figures

1 Introduction

2 Background

2.1 Interrupt-Driven Systems

An interrupt-driven system is an architectur where the flow of the execution is changed by unpredictable events in the system also known as interrupts. Interrupts can be caused by hardware devices, software conditions or external signals forcing the proscessor to suspend the current task to execute an interrupt handler or interrupt service routine(ISR). Interrupt-driven systems are used in real-time operating systems, embedded systems and generally in systems where timely responses are nessesary [1].

The management of the interrupts to keep the fast responsivness of the system is the challenging part of an interrupt-driven system. Interrupts occur unpredictable so you have to consider every execution flow. To ensure the execution of critical interrupts, interrupts are often prioritized, so higher priority events can interrupt lower ones and get handled immediatly. When handling an interrupt the current state of the processos is saved and the context is switched to the ISR [1].

There are different types of interrupts based on the source they come from. There are Hardware interrupts, software interrupts and external interrupts. Each interrupt has different mechnisms and priorities, which influence the behavior of the system to the interrupt [4].

The unpredictivness and asynchronous nature of the interrupts bring a lot of challenges in designing and implementing an interrupt-driven system. One of the biggest challenges is the correct handling of high-priority interrupts without delaying them substantually. Which needs an sophisticated scheduling and prioritization mechanism. The execution of main programm and ISR needs to be handled properly to ensure data intergrity. Furthermore, handling context switches, preserving system state and avoiding deadlocks complitcate the development of an interrupt-driven system [5].

2.2 Shared Resources

Shared resources are data or hardware components that can be accessed by multiple threads or processes in a concurrent system. These resources include memory locations, files, I/O devices, and communication channels. 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 avoid conflicts [8].

2.2.1 Types of Shared Resources

- `Memory`: Shared memory locations, such as global variables or heap-allocated data structures, are accessed by multiple threads. Proper synchronization mechanisms, like locks or atomic operations, are required to ensure that memory access is controlled and consistent [8].

- `:I/O Devices`: Hardware devices, such as printers, disk drives, and network interfaces, can be shared among different processes or threads. Access to these devices must be coordinated to prevent conflicts and ensure that operations are performed correctly [4].
- `Files`: Files on a disk can be accessed by multiple processes. File locks or similar mechanisms are used to manage concurrent access, ensuring that read and write operations do not interfere with each other [5].
- `Communication Channels`: Pipes, message queues, and shared memory segments used for inter-process communication are shared resources that require careful management to avoid data races and ensure the integrity of the communicated data [8]

2.2.2 Managing Shared Resources

Proper management of shared resources involves the use of synchronization mechanisms to coordinate access and ensure data consistency. Mutexes, semaphores, and condition variables are common tools used to control access to shared resources. Mutexes provide mutual exclusion, ensuring that only one thread can access the resource at a time. Semaphores can limit the number of threads accessing the resource simultaneously. Condition variables allow threads to wait for certain conditions to be met before proceeding, facilitating complex synchronization scenarios [8].

2.3 Reduced Inter-Procedural Control Flow Graphs (RICFG)

Control Flow Graphs (CFG) are representations of all possible path through a programm or a function during its execution. An Inter-Procedural Control Flow Graph (ICFG) adds possible edges between multiple programms or functions to also show possible control flows between those. A Reduced Inter-Procedural Control Flow Graph (RICFG) is an optimized version of the ICFG which simplifys the the graph to only the necessary informations needed for the analysis [2].

There are multiple techniques to reduce the graph, such as node merging, edge contraction and eliminaition of non-important nodes, without losing any informations required for the analysis and to reduce 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 but retaining enough detail, RICFG are great to use for static analysis of data races [1].

There are several techniques to construct a RICFG. Node merging is combining nodes that represent redundant control flow paths, to reduce the number of nodes in the graph. Edge contraction is simplifying the graph by reducing the amount of edges between nodes. It collapses edges, that do not significally affect the control flow of the graph. The elimination of nodes is the main tool used in this work to reduce the CFG. Eliminating nodes, which do not carry any essential information for the applied data

analysis significally reduces the amount of data the algorithm has to analyze. Overall these techniques enhance the scalability of the static analysis and make it more practical to analyze more complex data [6].

2.4 Data Races

Data races can occur when two or more functions or threads access a shared resource concurrently and also one of those accesses has to be a write opperation. This can lead to unpredictable behavior and errors in the system. This makes the detection of data races a critical aspect of concurrent programming [3].

Without proper synchronization a system with multiple threads or functions which use shared data lead to data races. The outcome of a program with data races is non-deterministic. The order of execution of operations can vary, leading to not reproducable or hard-to-reproduce bugs.

2.4.1 Types of Data Races

There are primarily two types of data races. Two write operations lead to unpredictable data corruption since the value of the shared resource depends on the order of execution of the two write operations. Data races with one read operation and one write operation can cause faulse programm behavior since the data that is read can be inconsistent due to the possibility of the concurrent write operation [3].

2.4.2 Detection Techniques

Static analysis examines the code without executing it. Static analysis tools analyze the source code to detect data races by simulating different execution paths of the program. It can identify potential data races by examining the control flow and data dependencies of shared data in the code. This enables early detection of possible issues.

Dynamic analysis, on the other hand, monitors the program during execution to identify potential race conditions. FastTrack [3] is a dynamic analysis tool that efficiently detects data races by maintaining a happens-before relationship among the threads' operations and checking for violations. Dynamic analysis provides a runtime perspective, capturing actual execution scenarios and detecting data races that may not be evident through static analysis alone.

Hybrid techniques combine static and dynamic analysis to leverage the strengths of both approaches. Static analysis can identify potential race conditions, which can then be confirmed or refuted by dynamic analysis during actual program execution. This approach allows for comprehensive detection and resolution of data races, ensuring both theoretical and practical coverage of potential issues.

2.4.3 Implications of Data Races

The presence of data races in a program can lead to several critical issues. The outcome of a program with data races is non-deterministic, making it difficult to reproduce and debug errors. Concurrent writes to shared data without proper synchronization can lead to data corruption, resulting in incorrect program behavior and potentially causing system crashes. Furthermore, data races can be exploited by malicious actors to create security vulnerabilities. Inconsistent data states can be manipulated to bypass security checks or corrupt sensitive data, posing significant risks to system integrity and security [7].

2.4.4 Strategies for Preventing Data Races

Preventing data races requires careful design and implementation of concurrent programs. One effective strategy is to use proper synchronization mechanisms, such as mutexes, semaphores, and condition variables, to control access to shared data. These mechanisms ensure that only one thread can access the shared data at a time, preventing conflicting operations. Avoiding shared mutable state is another effective strategy, where threads operate on local copies of data instead of shared data, reducing the potential for conflicts. Designing thread-safe data structures and algorithms that inherently manage concurrent access also helps prevent data races, ensuring reliable and predictable program behavior [8].

2.5 Static Detection of Data Races in Interrupt-Driven Systems

In interrupt-driven systems, the asynchronous nature of interrupts and the concurrent execution of ISRs and the main program introduce significant challenges in ensuring data consistency and detecting data races. Static analysis techniques, particularly those using RICFGs, offer a promising approach to identifying potential data races without the need for exhaustive testing or runtime monitoring [1].

The approach involves constructing RICFGs for the program, including both the main code and ISRs, capturing the control flow and potential interactions between them. By analyzing the RICFGs, paths where shared data is accessed concurrently without proper synchronization can be identified, indicating potential data races. Integrating the static analysis tool with the development workflow enables continuous detection and resolution of data races during the software development lifecycle, improving the reliability and correctness of interrupt-driven systems [1].

The methodology for static detection of data races in interrupt-driven systems involves several key steps. First, the RICFGs are constructed for the entire program, including the main code and ISRs. This involves analyzing the control flow and identifying points where the main program and ISRs interact with shared data. Next, the RICFGs are analyzed to identify potential race conditions, focusing on paths where concurrent access to shared data may occur without proper synchronization. Finally, the analysis results

are integrated with the development workflow, allowing developers to address identified race conditions early in the development process. This methodology ensures comprehensive coverage of potential race conditions and facilitates timely resolution of issues [1].

3 Implementation

Class BasicBlock

```
    class BasicBlock:
    def __init__(self, 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__(self):
    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)))
```

The class `BasicBlock` represents a basic block in a control flow graph. Each block has the following attributes:

- `function_name`: The name of the function to which the block belongs.
- `number`: The number of the basic block.
- `shared_resources`: A list of the shared resources used in the block.
- `successors`: A list of the successor blocks.
- `enable_disable_calls`: A list of calls to enable/disable ISRs within the block.
- `code`: The lines of code of the block.

Function parse_basic_blocks

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

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

The function `parse_basic_blocks` starts with the initialisation of a dictionary `blocks` and a variable `current_function`. It opens the file and reads its lines into a list `lines`.

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

These variables are initialised to collect information on basic blocks:

- `bb_num`: Number of the current basic block.
- `shared_resources`: List of shared resources in the current block.
- `enable_disable_calls`: List of ISR activation/deactivation calls.
- `code_lines`: The lines of code of the current block.
- `line_number`: The current line number in the file.

```
    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
```

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

```
    bb_match = re.match(r'<bb (\d+)>:', line)
    if bb_match:
    if bb_num is not None and current_function is not None:
    blocks[(current_function, bb_num)] = BasicBlock(
    current_function, bb_num, shared_resources, [], enable_disable_calls,
        code_lines)
    bb_num = int(bb_match.group(1))
    shared_resources = []
    enable_disable_calls = []
    code_lines = []
```

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

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

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

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

    code_lines.append((line, line_number))
```

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

```
    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)
```

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

```
    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
```

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

Function track_isr_status

```
    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
```

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

Function extract_isr_index

```
    def extract_isr_index(function_name):
    match = re.search(r'isr[_]?(\d+)', function_name)
    if match:
    return int(match.group(1)) - 1
    return None
```

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

Function detect_data_races

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

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

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

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

```

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

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

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

    process_block(block, current_isr_status)

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

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

```
    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)
        ])
```

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

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

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

```
check_for_data_races()
```

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

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

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

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

relevant_isr_disabled1 = is_isr_disabled(isr_status1, func2) and not
    is_isr_enabled_by_another(isr_status1, func2)
relevant_isr_disabled2 = is_isr_disabled(isr_status2, func1) and not
    is_isr_enabled_by_another(isr_status2, func1)

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

return filtered_data_races

filtered_data_races = filter_data_races(potential_data_races)

return filtered_data_races
```

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

4 Evaluation

1,5 Woche

5 Conclusion

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

Bibliography

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

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

Bibliography

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

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

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

[4] Burns, A., & Wellings, A. (2009). Real-Time Systems and Programming Languages. Addison-Wesley.

[5] Labrosse, J. J. (2002). MicroC/OS-II: The Real-Time Kernel. CMP Books.

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

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

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

Attachments

```
    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):
```

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

    return filtered_data_races

    filtered_data_races = filter_data_races(potential_data_races)

    return filtered_data_races


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

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

    data_races = detect_data_races(blocks)

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