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 30, 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 Background

1.1 Interrupt-Driven Systems

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

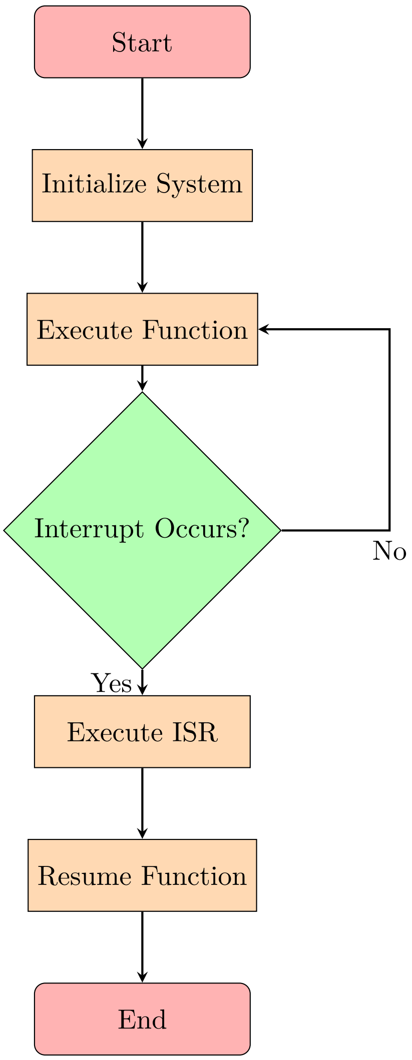

Figure 1.1: Flowchart of the interrupt-driven system

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

Managing the interrupts to maintain the fast responsiveness of the system is the most challenging part of an interrupt-driven system. Interrupts occur in unpredictable ways, so you have to consider every possible execution flow. To ensure the execution of critical interrupts, interrupts are often 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 the interrupts present many challenges in designing and implementing an interrupt-driven system. One of the biggest challenges is the correct handling of high-priority interrupts without delaying them substantially, which requires a sophisticated scheduling and prioritization mechanism. The execution of the main program and ISR needs to be handled properly to ensure data integrity. Furthermore, handling context switches, preserving system state, and avoiding deadlocks all contribute to the development of an interrupt-driven system.

1.2 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].

1.2.1 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 [**?**].

- `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 [**?**].

- `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].

1.2.2 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].

In interrupt-driven software, the synchronization of the shared resources often involves disabling and enabling interrupts [3]. Analyzing the management of the shared resources is a large part of the data race analysis, which is further explained later.

1.3 Reduced Inter-Procedural Control Flow Graphs (RICFG)

Control Flow Graphs (CFG) are representations of all possible paths through a program or a function during its execution. An Inter-Procedural Control Flow Graph (ICFG) adds possible edges between multiple programs 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 that simplifies the graph to only the necessary information needed for the analysis [2].

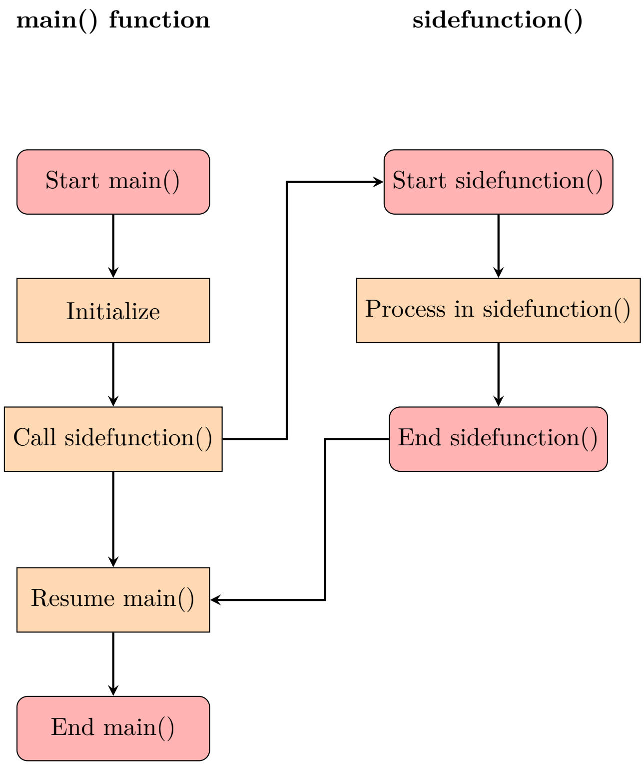

Figure 1.2: Example of Inter-Procedural Control Flow Graph

In Figure 2.2, a simple ICFG is shown. There are two separate linear control flow graphs where the main function calls the 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.

There are multiple techniques to reduce the graph, such as node merging, edge contraction, and the elimination of non-important nodes, without losing any information required for the analysis and reducing the complexity of the RICFG. The reduction of the ICFG makes the analysis of large and complex software much more efficient. By minimizing the amount of data while retaining enough detail, RICFGs are great for static analysis of data races [1].

Node merging is combining nodes that represent redundant control flow paths to reduce the number of nodes in the graph. Edge contraction is simplifying the graph by reducing the number of edges between nodes. It collapses edges that do not significantly affect the control flow of the graph [6]. The elimination of nodes is the main tool used in this work to reduce the CFG. Eliminating nodes that do not carry any essential information for the applied data analysis significantly reduces the amount of data the algorithm has to analyze. Overall, these techniques enhance the scalability of static analysis and make it more practical to analyze more complex data [1].

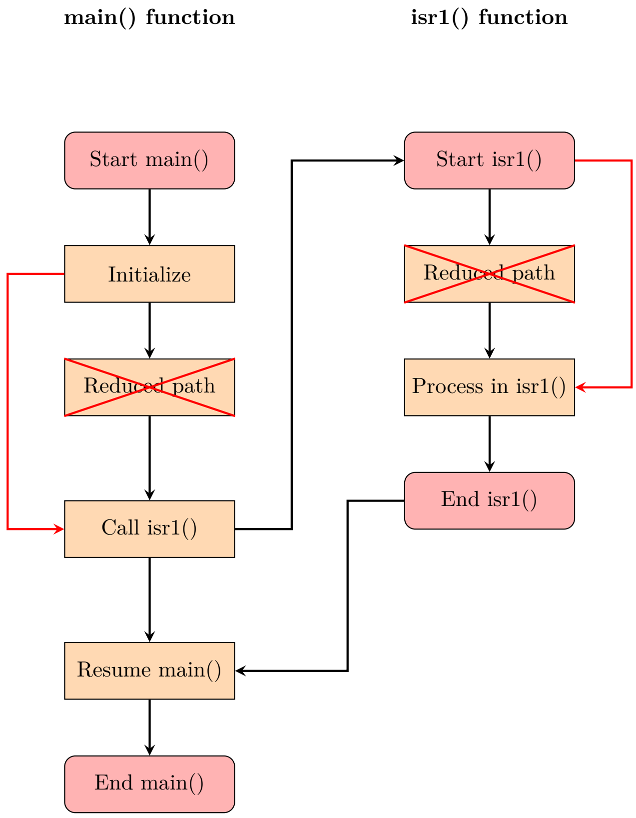

Figure 1.3: Example of Reduced Inter-Procedural Control Flow Graph

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

1.4 Data Races

A data race occurs when two or more functions or threads access a shared resource concurrently, without being ordered by a happens-before relationship, and one of those accesses is a write operation [5]. This can lead to unpredictable behavior and errors in the system, which makes the detection of data races a critical aspect of concurrent programs. Without proper synchronization, a system with multiple threads or functions that use shared data will lead to data races. The outcome of a program with data races is non-deterministic [5]. The order of execution of operations can vary, which may result in the generation of bugs that are not reproducible or difficult to reproduce.

Algorithm 1: Data Race Example

```
   Data: long shared1;
1  Function main ():
2      Variables:
3      unsigned char tmp;
4      Code:
5      tmp ← shared1;
6  Function isr1 ():
7      Code:
8      idlerun();
9      shared1 ← 1;
10     idlerun();
```

Figure 1.4: Simple Example of a Data Race

In Figure 2.4, an example of a simple data race is shown. A global variable shared1 is initialized 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.

1.4.1 Detection Techniques

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

Static Data Race Detection [1]

Advantages:

- Comprehensiveness: Static analysis inspects the code without executing the program by analyzing every possible execution path and interaction that could lead to data races.
- Early Detection: Since static analysis does not require execution, it can analyze the code in the development phase, allowing the developer to find issues without deployment.

Disadvantages:

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

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

Dynamic Data Race Detection [4]

Advantages:

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

Disadvantages:

- Performance Overhead: The analysis in runtime can slow down the application significantly.
- Coverage: The effectiveness is heavily dependent on the execution path triggered during the tests. If certain parts of the program are not executed during the test run, they are not analyzed.

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

1.4.2 Strategies for Preventing Data Races

Preventing data races requires careful design and implementation of concurrent programs. One effective strategy is to use proper 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 states is another effective strategy, where threads operate on local copies of data instead of shared data, reducing the potential for conflicts. Designing thread-safe data structures and algorithms that inherently manage concurrent access also helps prevent data races, ensuring reliable and predictable program behavior [8].

Preventing data races requires careful design and implementation of concurrent programs. Effective strategies for general prevention of data races are synchronization mechanisms such as mutexes, semaphores, and condition variables, which control access to shared data. These mechanisms ensure that only one thread can access the shared resource at a time [8]. Since I am focusing on data races in interrupt-driven systems, the main tool to prevent data races is to disable ISRs, which access shared resources in critical areas.

Algorithm 2: Enable/Disable ISR Call Example

```
   Data: long shared1
1  Function main():
2      Variables:
3      unsigned char tmp;
4      Code:
5      disable_isr(1);
6      tmp ← shared1;
7      enable_isr(1);
8  Function isr1():
9      Code:
10     idlerun();
11     shared1 ← 1;
12     idlerun();
13 Function isr2():
14     Code:
15     idlerun();
16     int variable1 = 1;
17     idlerun();
```

Figure 1.5: Example of a Data Race with Enable/Disable ISR Calls

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

1.5 Static Detection of Data Races in Interrupt-Driven Systems

The asynchronous nature and concurrent execution of ISRs and the main function introduce significant challenges for data consistency and detecting data races in interrupt-driven systems. Static data race analysis, especially those using RICFGs, is a promising approach to identifying data races without the need for extensive testing and runtime monitoring as in dynamic approaches [1].

The static approach involves the construction of an RICFG for the program, which includes both the main code and ISRs, and capturing the control flow and potential interaction between them. Analyzing the RICFG shows paths where shared resources are accessed concurrently without proper synchronization and indicates potential data races. Integrating the static analysis tool with the development process enables continuous detection of data races during software development, improving the reliability and correctness of interrupt-driven systems [1].

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

Algorithm 3: Static Race Detection

Input: RICFGs of P

Output: potential racing pairs (PR)

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

Figure 1.6: Static Race Detection Approach by [1]

The approach by Wang et al. shows a computation of potential data races using RICFGs. By running a depth-first search on the RICFGs it finds the interrupt status of every instruction. If there is a shared resource in both of the analyzed RICFGs, at least one of them is a write operation, and the two functions differ in their priority. While the interrupt in this pair is enabled, the two accesses are a potential data race [1]. In the following, I am going to introduce the implementation of my static analysis program based on the static race detection approach of Wang et al.

2 Implementation

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

Class BasicBlock

Algorithm 4: BasicBlock Class Definition

```
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
```

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

- `function_name`: The function name to which the basic block belongs.
- `number`: The number of the basic block.
- `shared_resources`: All accesses of shared resources within the basic block. The access type (read/write) and also the line number of such calls are saved.
- `successors`: A list of all the successors of each basic block. Important to build all possible paths through the CFG.
- `enable_disable_calls`: All calls that disable or enable an ISR within this basic block and also with the corrsponding line number of those calls to ensure correct order.
- `code`: The lines of code of the block stored with their line number.

Parsing

Algorithm 5: Parse Basic Blocks

```
Data: file_path, shared_resource_names
Result: Dictionary of BasicBlocks mapping (function name, block number) to
        BasicBlock instances
blocks ← empty dictionary
current_function ← None
Input: Path to source file, list of shared resource names
Output: Dictionary of basic blocks with details
for each line in file at file_path do
    line ← line.strip()
    Increment line_number
    if line starts new function then
        if current block is open then
            Save current block to blocks
        end
        Update current_function and reset current block details
    end
    else if line starts new basic block then
        if current block is open then
            Save current block to blocks
        end
        Initialize new block details
    end
    Update shared_resources and enable_disable_calls based on line content
end
if current block is open at end of file then
    Save last block to blocks
end
foreach line indicating successor blocks do
    Parse successor details and update block information in blocks
end
return blocks
```

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

These variables are initialised to collect information on basic blocks:

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

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

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

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

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

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

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

Algorithm 6: Track ISR Status

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

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

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

Algorithm 7: Extract ISR Index

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

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

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

Data Race Detection

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

Algorithm 8: Initialization and ISR Enabling Map

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

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

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

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

Algorithm 9: Process Block

```
Data: block, current_isr_status
Result: Updated ISR status and recorded resource accesses
Input: A code block and the current ISR status as a list
Output: Updated current ISR status and appended resource accesses to a global
        list
foreach line, line_number in block.code do
    if line contains 'enable_isr' or 'disable_isr' then
        isr_idx_match ← search for pattern '+' in line
        if isr_idx_match is found then
            isr_idx ← integer value of the first group in isr_idx_match minus one
            if "disable_isr" is in line then
                if 0 ≤ isr_idx < length of current_isr_status then
                    current_isr_status[isr_idx] ← 1
                end
            else
                if 0 ≤ isr_idx < length of current_isr_status then
                    current_isr_status[isr_idx] ← 0
                end
            end
        end
    end
    foreach resource_name, access_type, res_line_number in
      block.shared_resources do
        if res_line_number == line_number then
            resource_accesses[resource_name].append((block.function_name,
              block.number, access_type, line_number,
              current_isr_status.copy()))
        end
    end
end
```

This part of the code runs thorugh each line of an basic block to find the current ISR status while resources get accessed. When a resource is found all the informations are added to the `resource_accesses` dictionary which include the function name and the block number of the current basic block aswell as the access type, the line number and the ISR status of the access. All those informations are used later for the detection of the data races.

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

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

```
if (block.function_name, block.number) is in visited_blocks then
    return
end
visited_blocks.add((block.function_name, block.number))
path.append((block.function_name, block.number))
process_block(block, current_isr_status)
if block has successors then
    foreach successor in block.successors do
        dfs(successor, set(visited_blocks), current_isr_status.copy(), path.copy())
    end
end
```

The `Depth-First Search (DFS)` function is recursively processing each

Algorithm 11: Initial Processing and Recursive Depth-First Search

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

```
foreach (func_name, bb_num) and block in blocks do
    if bb_num = 2 then
        initial_isr_status ← track_isr_status(blocks)
        process_block(block, initial_isr_status)
        foreach successor in block.successors do
            dfs(successor, {}, initial_isr_status.copy(), [(func_name, bb_num)])
        end
    end
end
```

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

Algorithm 12: Check for Data Races

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

```
foreach resource, accesses in resource_accesses do
    for i ← 0 to length(accesses)-1 do
        func1, bb_num1, access_type1, line_number1, isr_status1 ← accesses[i]
        for j ← i+1 to length(accesses)-1 do
            func2, bb_num2, access_type2, line_number2, isr_status2 ←
             accesses[j]
            if func1 ≠ func2 and (access_type1 = "write" or access_type2 =
             "write") then
                potential_data_races.append((resource, (func1, bb_num1,
                 access_type1, line_number1, isr_status1), (func2, bb_num2,
                 access_type2, line_number2, isr_status2)))
            end
        end
    end
end
```

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

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

Filter Possible Data Races

Algorithm 13: Filter Data Races

```
Data: potential_data_races
Result: Filtered list of data races
Function filter_data_races(potential_data_races):
    filtered_data_races ← empty list
    foreach resource, access1, access2 in potential_data_races do
        Decompose access1 into func1, bb_num1, access_type1, line_number1,
          isr_status1
        Decompose access2 into func2, bb_num2, access_type2, line_number2,
          isr_status2
        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) then
            filtered_data_races.append((resource, access1, access2))
        end
    end
    return filtered_data_races
end
Function is_isr_disabled(isr_status, func_name):
    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
        // 1 indicates disabled
    end
    return False
end
Function is_isr_enabled_by_another(isr_status, func_name):
    isr_idx ← extract_isr_index(func_name)
    if isr_idx is not None then
        foreach enabler_isr, enabled_isrs in isr_enabling_map 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
end
```

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

3 Evaluation

1,5 Woche

4 Conclusion

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

Bibliography

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

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

Bibliography

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

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

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

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

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

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

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

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

Attachments

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

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

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

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

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

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

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

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

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

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

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

code_lines.append((line, line_number))

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

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

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

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

return blocks

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

def extract_isr_index(function_name):
```

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

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

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

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


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

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

process_block(block, current_isr_status)

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

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


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

check_for_data_races()


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

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

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

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

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

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

        return filtered_data_races

        filtered_data_races = filter_data_races(potential_data_races)

        return filtered_data_races


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

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

        data_races = detect_data_races(blocks)

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