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 26, 2024

supervised by:
Prof. Dr. Sibylle Schupp
Ulrike Engeln

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

STS Software Technology Systems

Contents

List of Figures

1 Introduction

2 Background

2.1 Interrupt-Driven Systems

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

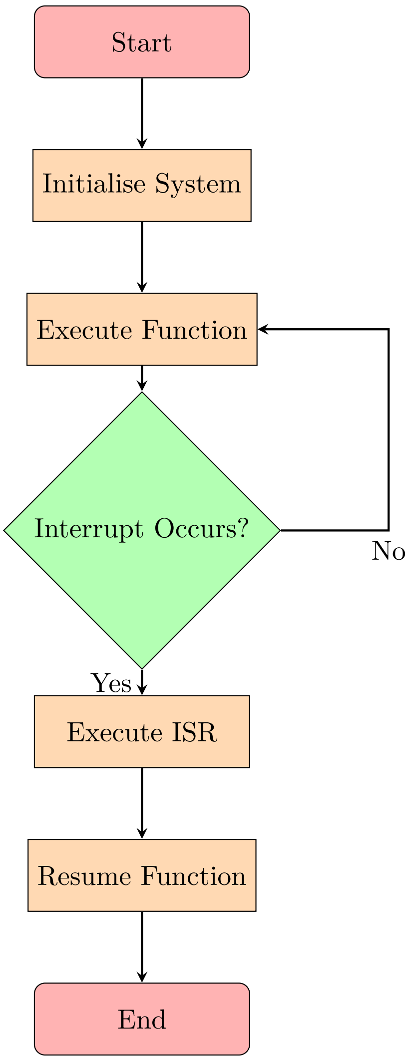

Figure 2.1: Flow-Chart des interruptgesteuerten Systems

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

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

The unpredictivity and asynchronous nature of the interrupts present 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 substantially. Which needs a sophisticated scheduling and prioritisation mechanism. The execution of the main programme and ISR needs to be handled properly to ensure data tegrity. Furthermore, handling context switches, preserving system state, and avoiding deadlocks all contribute to the development of an interrupt-driven system.

2.2 Shared Resources

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 synchronisation problems. In interrupt-driven systems, shared resources often involve variables or data structures that are accessed by both the main programme and ISRs. Proper management of shared resources is critical to ensuring data consistency and avoiding conflicts [8]. Proper management of shared resources involves the use of synchronisation 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 synchronisation scenarios [8]. In interrupt driven software, the synchronisation of the shared resources often implies disabling-enabling interrupts [3]. Analysing the management of the shared resources is a large part of the data race analysis, which is further explained later.

2.3 Reduced Inter-Procedural Control Flow Graphs (RICFG)

Control Flow Graphs (CFG) are representations of all possible paths through a programme or a function during its execution. An Inter-Procedural Control Flow Graph (ICFG) adds possible edges between multiple programmes or functions to also show possible control flows between those. A Reduced Inter-Procedural Control Flow Graph

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

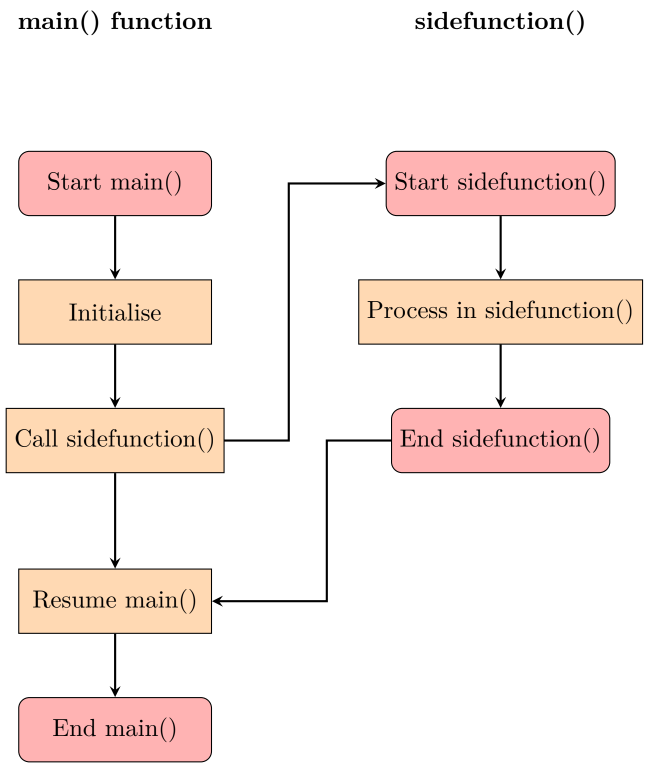

Figure 2.2: Example of Inter-Procedural Control Flow Graph

In Figure 2.2, a simple ICFG is shown. There are two separate linear control flow graphs where the main function calls the sidefunction in its execution. To interpret the flow of the programme 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 a lot more efficient. By minimising 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 analyse more complex data [1].

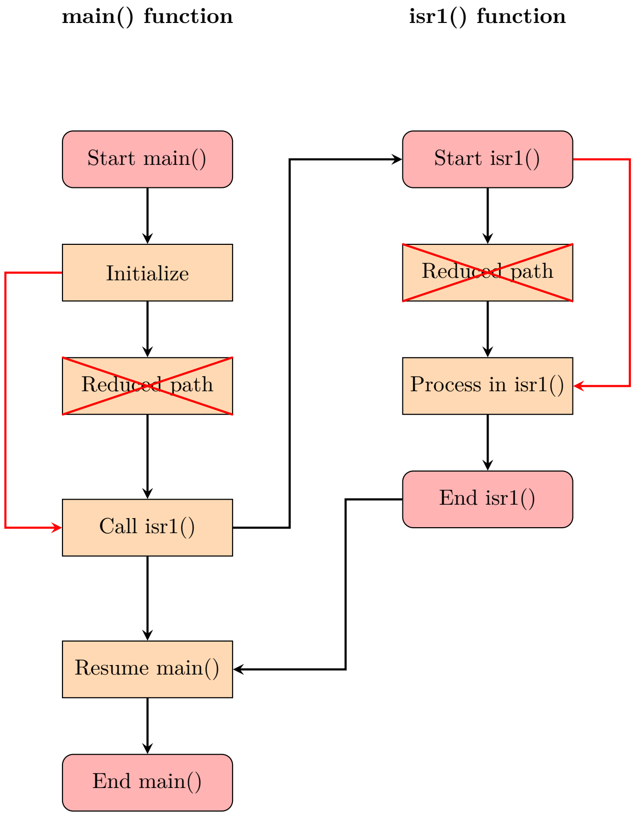

Figure 2.3: Example of Reduced Inter-Procedural Control Flow Graph

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.

2.4 Data Races

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

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

Figure 2.4: Simple Example of a Data Race

In Figure 2.4, an example of a simple data race is shown. A global variable shared1 is initiated and accessed in two different functions main() and isr1 (). Since there are no synchronisation 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 those methods provides benefits and challenges.

Static Data Race Detection [1]

Advantages:

- Comprehensivness: Static analysis inspects the code without executing the programme by analysing every possible execution path and interactions that could lead to data races.
- Early Detection: Since static analysis does not require execution, it can analyse 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 Behaviour: Dynamic behaviours such as pointers or recursion can be challenging to analyse 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 programme, identifying data races in real-time. Which results in reducing the number of false positives.
- Context-Sensitive Detection: By analysing the actual runtime behaviour, 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 programme are not passed through in the execution run, they are not analysed.

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

2.4.2 Strategies for Preventing Data Races

Preventing data races requires careful design and implementation of concurrent programs. One effective strategy is to use proper synchronisation 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 programme behaviour [8].

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

Algorithm 2: Enable/Disable ISR Call Example

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

Figure 2.5: 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 save access of the shared data. The main function and isr1 both access the shared resource shared1. Since the read operation in line 6 of the main function is safely accessed by disabling isr1 in line 5 and enabling it in line 7, a possible data race is prevented.

2.5 Static Detection of Data Races in Interrupt-Driven Systems

The asynchronous nature and concurrent execution of ISRs and the main function introduces significant challenges for data consistency and detecting data races in interrupt-driven systems. Static data race analysis, especially those using RICFGs, are 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 programme, which includes both the main code and ISRs, and capturing the control flow and potential interaction between them. Analysing the RICFG, shows paths where shared resources are accessed concurrently without proper synchronisation 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 programme, including the main code and the ISRs. This involves analysing the control flow and identifying interactions between the main programme and ISRs. Next, the RICFGs are analysed to find potential data races, focusing on paths where concurrent access of 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 2.6: 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 analysed RICFGs, at least one of them is a write operation, and the two functions differ in their priority. While the interrupt in this pair is enabled, the two accesses are a potential data race [1]. In the following, I am going to introduce you to the implementation of my static analysis programme based on the static race detection approach of Wang et al..

3 Implementation

4 Evaluation

5 Conclusion

Bibliography

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

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

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

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

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

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

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

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

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()
```