

ب نام آنک جان را فکرت آموخت

امیر ارسلان یآوری - ۴۰۲۲۰۳۴۹۷

امتحان درس سیستم‌های تحمل‌پذیر اشکال دانشگاه صنعتی شریف استاد: دکتر اجلائی
پاسخ سوال دوم نیمسال اول ۱۴۰۴-۱۴۰۳

۲) یک روش کشف خطای CFE که می‌توان آن را در سطح کد منبع (source code) به جای سطح زبان ماشین (زبان اسمبلی) اعمال نمود به این شکل است که برای تمام توابع (روتین‌ها یا متودهای) موجود در یک برنامه در ابتدا که وارد آن تابع می‌شویم id خاص آن تابع را در یک پشته push می‌کنیم و سپس هر زمان که هنگام بازگشت از آن تابع است همان id را از پشته pop می‌کنیم و واری می‌کنیم که آیا مقدار id همخوانی دارد یا خیر. به این ترتیب اگر خروج نادرستی از یک تابع (روتین یا متود) وجود داشته باشد، یا ورود اشتباهی به میانه‌ی آن وجود داشته باشد و یا حتی اگر پشته دچار اشکال گردد توسط این روش کشف می‌شود. اعمال چنین روشی بصورت دستی (manual) توسط توسعه دهنده‌ی کد کار آسانی نبوده و خود دارای احتمال بالای خطای انسانی (خطای طراحی یا خطای پیاده‌سازی) است. روش صحیح اعمال چنین روش‌هایی این است که اعمال آن‌ها خودکار گردد. یک قطعه کد به زبان Python بنویسید که برای قطعه‌کدهای نوشته شده به زبان C++ این کار را انجام دهد. به این شکل که کد Python شما کد اولیه به زبان C++ را دریافت می‌کند و سپس یک کد C++ جدید تولید می‌کند که روش کشف خطای مذکور به آن اعمال شده است.

برای حل این سوال فرض اولیه این است که کد به حالت تمیز نوشته است و کد CPP به حالت ناخوانا و تک خطی نوشته نشده است. در صورتی که این فرض برای کد صادق نیست لطفاً در ابتدا از یک Linter یا Formatter استفاده نموده و کد را به حالت تمیز و خوانا ببرید. لازم به ذکر است که لازم است دستور clang-format را نیز داشته باشید. در غیر اینصورت خط مربوطه را از فایل پایتون پاک کرده و کد CPP را به حالت Inline Braces ببرید.

همچنین من دو فایل تست که خودم با آنها کدم را تست کردم با عنوان example1.cpp و example2.cpp در کنار این فایل قرار داده‌ام. در ادامه به توضیح کد خواهم پرداخت.

توضیح کد:

در این کد من ابتدا از clang-format استفاده کردم تا فرمت کد را مطابق با استاندارد Clang منطبق کنم. سپس یک تابع `process_cpp` را بر روی فایل CPP صدا زده و `newline` های اضافی را پاک می‌کنم. بعد از موارد ذکر شده یک تابع با نام `makes_fault_tolerant` فراخوانی می‌کنم که عملیات اضافه کردن کد `stack` به کد CPP در آن اتفاق می‌افتد که در ادامه آن را مفصلاً توضیح خواهم داد. لازم به ذکر است کد پایتون را باید به فرمت زیر اجرا نمایید.

```
python Q2.py <target_file.cpp> <output_file.cpp>
```

(clang-format و تابع `preprocess_cpp` بر روی کد اصلی اجرا می‌شوند چون تمیز شدن آن کد منطقی است).

یک فرض اصلی هم که وجود دارد و از استاد پرسیدم این است که فرض داریم یک فایل `cpp` داریم و برنامه به صورت `multi file` نوشته نشده است.

```
17 def makes_fault_tolerant(input_file, output_file):
18     flag = 0
19     with open(input_file, 'r') as infile, open(output_file, 'w') as outfile:
20         lines = infile.readlines()
21         id_counter = 0
22         stack_name = 'id_stack'
23
24         outfile.write(f"#include <stack>\nstd::stack<int> {stack_name};\n\n")
25
26         inside_function = False
27         braces_count = 0
28         inside_multiline_comment = False
29
30         control_keywords = ['if', 'for', 'while', 'switch', 'catch']
31
32     for line in lines:
33         # Check for start of multi-line comment
34 > if '/*' in line and not inside_multiline_comment: ...
38
39         # Check for end of multi-line comment
40 > if '*/' in line and inside_multiline_comment: ...
44
45         # Skip entire line if inside multi-line comment
46 > if inside_multiline_comment: ...
49
50         stripped_line = line.strip()
51
52         # Skip single-line comments
53 > if re.match(r'//.*', stripped_line): ...
56
57         # Check if the line starts with any control keywords
58 > if any(re.match(rf'^\s*{keyword}\b', stripped_line) for keyword in control_keywords): ...
63
64 > if re.match(r'.*?\s*?(?!\s*?)\s*?.*?', stripped_line): ...
75
76 > if re.match(r'^(!.*=).*?\s*?.*', stripped_line): ...
84
85 > if inside_function: ...
107
108         outfile.write(line)
```

در ابتدای فانکشن فایل ورودی `CPP` بدون مکانیزم تحمل‌پذیری اشکال به حالت خواندنی و فایل خروجی به حالت نوشتنی باز خواهند شد. سپس در ابتدای فایل خروجی تابع `stack` به همراه یک استک با اسم `id_stack` که در برنامه مورد استفاده قرار خواهد گرفت و به صورت گلوبال تعریف می‌شود که در همه جای کد در دسترس باشد. سپس در ادامه تعدادی متغیر کنترلی تعریف می‌کنم که بدانم تعداد curly brace های خوانده شده چه تعداد بوده، آیا یک کامنت به صورت `/**` است و یا آیا داخل یک تابع هستیم یا خیر. همچنین یک متغیر `id_counter` دارم که برای هر تابع یک `id` خاص و غیر تکراری را در داخل خود نگه می‌دارد (از صفر شروع می‌شود و پس از هر بار استفاده از آن یکی به مقدارش اضافه می‌شود). لازم به ذکر است در مدل کدی که من زدم خطوط فایل `CPP` یک به یک خوانده می‌شوند و برای همین است که متغیرهای کنترلی استفاده کردم تا بدانم در هر لحظه در کجای کد قرار دارم.

پس از آن، شرط اول و دوم کامنت‌های چند خطی را شناسایی می‌کنند و شرط سوم مشخص می‌کند که در صورتی که با این مدل از خطوط (کامنت‌های چند خطی) در فایل `target.cpp` مواجه شویم هیچ کاری انجام نداده و فقط آن‌ها را در فایل `output.cpp` چاپ می‌کنیم. در ادامه خط را `stripe` می‌کنم تا کنترل بهتری روی آن داشته باشم و شرط چهارم هم برای کامنت‌های تک خطی که به صورت `//` هستند استفاده شده. اگر خط کامنتی ببینیم مجدداً آن را بدون هیچ تغییری در فایل

output.cpp چاپ می‌کنیم. شرط پنجم چک می‌کند که برای کلید واژه‌های 'if', 'for', 'while', 'switch' عملیات افزودن push و pop را رد کند چراکه تنها به توابع این خطوط باید اضافه شود.

در خطوط بعدی با استفاده از رجکس توابع یک خطی و چند خطی را تشخیص می‌دهیم و اگر تابع تک خطی داشتیم push و چک کردن و pop را به آن اضافه می‌کنیم. اگر هم تابع چند خطی داشتیم با استفاده از متغیر کنترلی inside_function متوجه این موضوع می‌شویم و قبل از return یا در صورت عدم وجود آن با استفاده از braces_count قبل از آخرین curly brace خطوط مورد نظر را اضافه می‌کنیم.

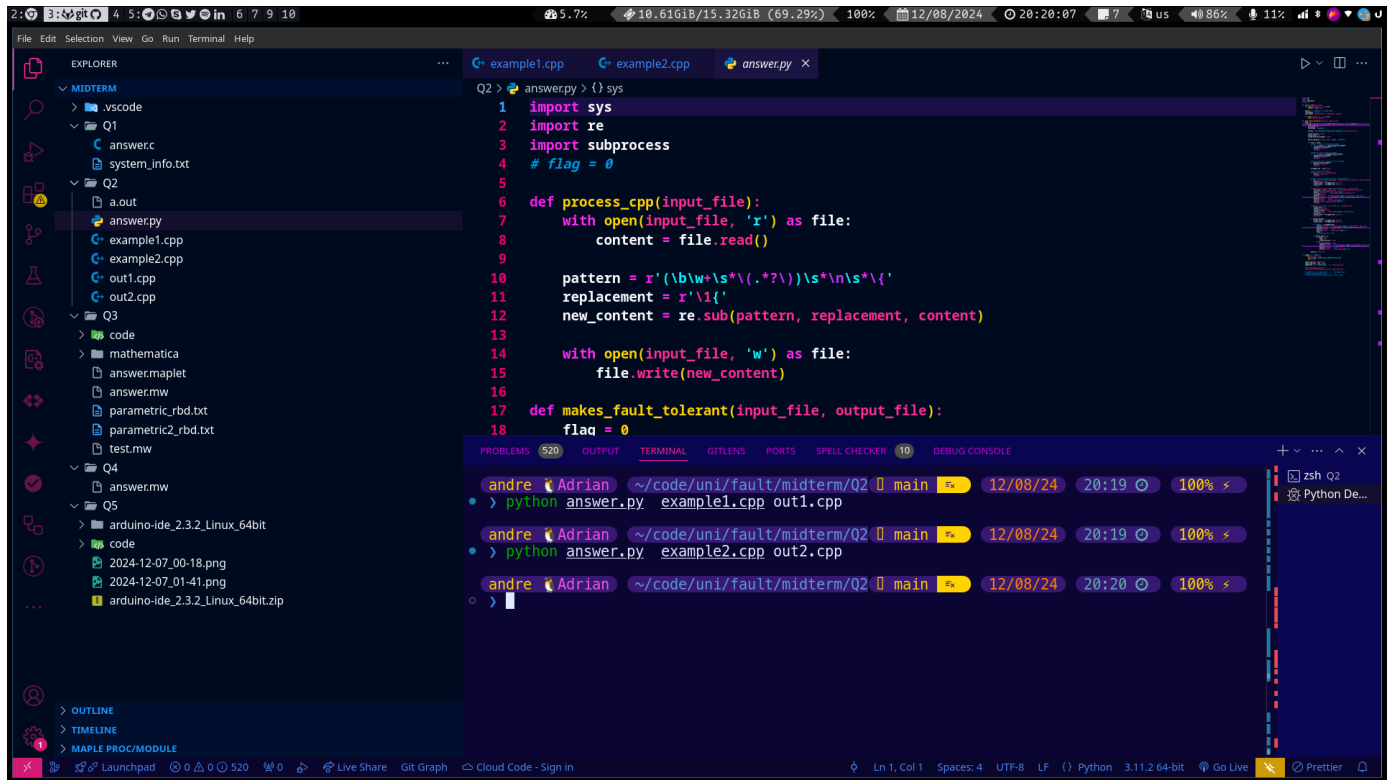
```
64         if re.match(r'.*?\s*?\s*\s*?\s*?\s*?', stripped_line):
65             last_index = len(stripped_line) - 1 - stripped_line[::-1].index('{')
66             outfile.write(stripped_line[:last_index])
67             outfile.write(f"\n    {stack_name}.push({id_counter});\n")
68             id_counter += 1
69             outfile.write(f"    if ({stack_name}.top() != {id_counter - 1}) {{\n")
70             outfile.write(f"        std::cerr << "Stack mismatch error!\n";\n")
71             outfile.write(f"    }}\n")
72             outfile.write(f"    {stack_name}.pop();\n")
73             outfile.write(stripped_line[last_index:])
74             continue
75
76 > if re.match(r'^(!.*=).*?\s*?\s*?', stripped_line): ...
84
85     if inside_function:
86         braces_count += stripped_line.count('{')
87         braces_count -= stripped_line.count('}')
88
89     if 'return' in stripped_line:
90         outfile.write(f"    if ({stack_name}.top() != {id_counter - 1}) {{\n")
91         outfile.write(f"        std::cerr << "Stack mismatch error!\n";\n")
92         outfile.write(f"    }}\n")
93         outfile.write(f"    {stack_name}.pop();\n")
94         flag = 1
95         # inside_function = False
96
97     if braces_count == 0:
98         if (flag == 1):
99             flag = 0
100             inside_function = False
101         else:
102             inside_function = False
103             outfile.write(f"    if ({stack_name}.top() != {id_counter - 1}) {{\n")
104             outfile.write(f"        std::cerr << "Stack mismatch error!\n";\n")
105             outfile.write(f"    }}\n")
106             outfile.write(f"    {stack_name}.pop();\n")
107
```

شروط آخر و دو تا مانده به آخر نمایش کاملی از نحوه‌ی اضافه کردن push، چک کردن استک و pop را نمایش می‌دهد.

منطق افزودن این کدها بدین صورت است که در اولین خط تابع push به استک با id منحصر به فرد آن انجام می‌شود و در آخرین خطوط تابع که از آن قرار است بیرون بیاییم اول top استک را آیدی منحصر به فرد تابع چک می‌کند اگر اشتباه بود ارور می‌دهد و اگر درست بود آن را از استک pop می‌کند که حجم زیاد و بیهوده‌ای هم برنامه‌ی ما اشغال نکند.

این کد بر روی یک سیستم لینوکسی X64 دبیان انجام شده.

نتایج اجرای کدها به عنوان نمونه در تصویر زیر آمده است؛ همچنین بررسی شده است که کد تغییر یافته قابل کامپایل و اجرا نیز باشد.



```
Q2 > python answer.py example1.cpp out1.cpp
Q2 > python answer.py example2.cpp out2.cpp
```

```
1 import sys
2 import re
3 import subprocess
4 # flag = 0
5
6 def process_cpp(input_file):
7     with open(input_file, 'r') as file:
8         content = file.read()
9
10    pattern = r'(\b#w\s*(.?.?))\s*\n\s*{'
11    replacement = r'\1{'
12    new_content = re.sub(pattern, replacement, content)
13
14    with open(input_file, 'w') as file:
15        file.write(new_content)
16
17 def makes_fault_tolerant(input_file, output_file):
18     flag = 0
```

قسمت‌های توضیح داده نشده قسمت‌های بدیهی کد بوده‌اند و دارای منطق پیاده‌سازی نبودند. برای مثال یک مورد از رجکس‌های استفاده شده را توضیح می‌دهم ولی باقی موارد همانند سینتکس کد قابل توضیح نیستند چرا که منطق آن توضیح داده شده است.

`r'.*?\\(\\s*?.*?\\s*?\\)\\s*?.*?{\\s*?}.*?'`

رجکس (Regex) یک الگوی متنی است که برای جستجو و پردازش رشته‌های متن استفاده می‌شود. در رجکس خط بالا تمامی خطوطی که ابتدای آنها صفر الی هر تعداد کاراکتر وجود دارد و بعد از آن یک کاراکتر (وجود دارد و بعدش صفر الی هر تعداد کاراکتر فاصله وجود داشته باشد و بعدش هر تعداد از صفر تا هر چندتا کاراکتری باشد و دوباره صفر الی هر تعداد کاراکتر فاصله وجود داشته که بعد آنها یک کاراکتر (وجود داشته باشد و بعد از آن به همین ترتیب تعدادی کاراکتر فاصله و هر کاراکتری باشد و یک کاراکتر { باشد و تعدادی space داشته باشد و یک } و در پایان هم تعدادی کاراکتر داشته باشد را شناسایی می‌کند. برای اطلاعات بیشتر می‌توانید به لینک زیر مراجعه فرمایید.

https://www.w3schools.com/python/python_regex.asp

(با تشکر :)