Faculty of Engineering and Applied Science

SOFE 3980U Software Quality

Winter 2022

Dr. Akramul Azim

TAs: Md Asif Khan, Rezwana Mamata, Alireza Saberironaghi

Assignment 2:
Static and Dynamic Analysis

Jessica Leishman
(100747155)

March 26, 2022

Table of Contents

GITHUB REPOSITORY:

https://github.com/jessica-leishman/high-rollers

Static Analysis:
https://github.com/jessica-leishman/high-rollers/tree/main/analysis_static

Dynamic Analysis:
https://github.com/jessica-leishman/high-rollers/tree/main/analysis_dynamic

Static Analysis: Program Slicing

Six variables were analyzed using the forward slicing technique on the highrollers.py source code. This slicing was conducted in both an automated and manual manner. Both the automated and manual techniques provided unique advantages which will be later discussed within this section.

In some cases, it was necessary to integrate some elements of backward slicing due to co-dependencies on a set of variables (control structures and updating of additional variables), however, this does not detract from the program slices created and instead assists with critical comprehension. On the contrary, a major detriment to the automated slicer was its tendency to include additional logic structures that were *not* relevant to the variable being sliced on.

Some further limitations of the automated slicer include: no support for multi-line if/else structures, it cannot support the use of different names for the same variables (requiring it to be rewritten prior to use with the slicer), comments must be removed, there is a lack of structure and clarity for where each snippet in the slice comes from, lines that contain the variable as part of another word will also be included even if not related (i.e. including variable “catPath” when looking for variable “cat”). Additionally, the output format of the slices contains all indentation from the source program, and the slicer is case sensitive when entering the variable to slice on.

Automated program slicing operates on the highrollers.txt file that must be prepared to the above limitations. This includes renaming variables to be consistent, and removing comments. It creates a list of all the lines in the text file, then iterates through them looking for the variable. If the variable is found, it checks if there are any if/elif/else statements in the line prior. If the line itself is a if/elif statement containing the variable, the line after it is also included. This allows for variable assignments related to the sliced variable and control flow structures to be included, even if not to the fullest extent of the original source material.

Automated Program Slice Screenshots

Slicing on variable score

```
PS D:\GitHub\high-rollers> python -u "d:\GitHub\high-rollers\analysis_static\automated\programslicer.py"
Enter the variable to parse for: score
Program slice on variable:  score

scoreFont = pygame.font.SysFont('rubik', 24)

def updateScore(score, state):
    if state == 0:
        score = (score-1)
    elif state == 1:
        score = (score+1)
    return score

def displayScore(score):
    score = str(score)
    text_on_screen('Score:', scoreFont, gameColours['dY'], screen, (width-90), 30)
    text_on_screen(score, scoreFont, gameColours['dY'], screen, (width-25), 30)

def gameTime(score):
    score = score
        displayScore(score)
            if click:
                gameLogic(score)

def gameLogic(score):
    score = score
        displayScore(score)
                score = updateScore(score, state)
                if state == 0:
                    loseScreen(die1, die2, state, score)
                elif state == 1:
                    winScreen(die1, die2, state, score)
                else:
                    drawScreen(die1, state, score)

def winScreen(die1, die2, state, score):
    score = score
        displayScore(score)
            if click:
                gameTime(score)
```

```
def loseScreen(die1, die2, state, score):
    score = score
        displayScore(score)
            if click:
                gameTime(score)

def drawScreen(die1, state, score):
    score = score
        displayScore(score)
            if click:
                gameTime(score)

PS D:\GitHub\high-rollers>
```

Figure 1: Program slice conducted using the programslicer.py file on variable “score”.

Slicing on variable state

```
PS D:\GitHub\high-rollers> python -u "d:\GitHub\high-rollers\analysis_static\automated\programslicer.py"
Enter the variable to parse for: state
Program slice on variable:  state


def getCatPath(state):
    if state == 0:
        catPath = "assets/wincat.png"

    elif state == 1:
        catPath = "assets/losecat.png"

    elif state == 2:
        catPath = "assets/draw.png"


def updateScore(score, state):
    if state == 0:
        score = (score-1)

    elif state == 1:
        score = (score+1)

            if ev.type == dust_clear_event:
                state = checkWinner(die1, die2)
                score = updateScore(score, state)
                if state == 0:
                    loseScreen(die1, die2, state, score)
                elif state == 1:
                    winScreen(die1, die2, state, score)
                else:
                    drawScreen(die1, state, score)

def winScreen(die1, die2, state, score):
    state = state
        catPath = getCatPath(state)

def loseScreen(die1, die2, state, score):
    state = state
        catPath = getCatPath(state)

def drawScreen(die1, state, score):
    state = state
        catPath = getCatPath(state)

PS D:\GitHub\high-rollers>
```

Figure 2: Program slice conducted using the programslicer.py file on variable “state”.

Slicing on variable tuple mx, my

```
PS D:\GitHub\high-rollers> python -u "d:\GitHub\high-rollers\analysis_static\automated\programslicer.py"
Enter the variable to parse for: mx, my
Program slice on variable:  mx, my

        mx, my = pygame.mouse.get_pos()
        if playButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dG'], playButton)

        mx, my = pygame.mouse.get_pos()
        if rollButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dR'], rollButton)

        mx, my = pygame.mouse.get_pos()
        if againButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dG'], againButton)

        if quitButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dR'], quitButton)

        mx, my = pygame.mouse.get_pos()
        if againButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dG'], againButton)

        if quitButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dR'], quitButton)

        mx, my = pygame.mouse.get_pos()
        if againButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dG'], againButton)

        if quitButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dR'], quitButton)


PS D:\GitHub\high-rollers>
```

Figure 3: Program slice conducted using the programslicer.py file on variable(s) “mx, my”. These variables are frequently used as a tuple.

Slicing on variable catPath

```
PS D:\GitHub\high-rollers> python -u "d:\GitHub\high-rollers\analysis_static\automated\programslicer.py"
Enter the variable to parse for: catPath
Program slice on variable:  catPath

    if state == 0:
        catPath = "assets/wincat.png"
    elif state == 1:
        catPath = "assets/losecat.png"
    elif state == 2:
        catPath = "assets/draw.png"
    return catPath
        catPath = getCatPath(state)
        cat = pygame.transform.scale(pygame.image.load(catPath).convert_alpha(), (400, 450))
        catPath = getCatPath(state)
        cat = pygame.transform.scale(pygame.image.load(catPath).convert_alpha(), (400, 450))
        catPath = getCatPath(state)
        cat = pygame.transform.scale(pygame.image.load(catPath).convert_alpha(), (400, 450))

PS D:\GitHub\high-rollers>
```

Figure 4: Program slice conducted using the programslicer.py file on variable “catPath”.

Slicing on variable die1 (Computer's Roll)

```
PS D:\GitHub\high-rollers> python -u "d:\GitHub\high-rollers\analysis_static\automated\programslicer.py"
Enter the variable to parse for: die1
Program slice on variable:  die1


def checkWinner(die1, die2):
    if (die1 > die2):
        return 0

    elif(die1 < die2):
        return 1

    elif(die1 == die2):
        return 2


def getDice(die1, die2):
    if die1 == 1:
        compRoll = "assets/b1.png"

    elif die1 == 2:
        compRoll = "assets/b2.png"

    elif die1 == 3:
        compRoll = "assets/b3.png"

    elif die1 == 4:
        compRoll = "assets/b4.png"

    elif die1 == 5:
        compRoll = "assets/b5.png"

    elif die1 == 6:
        compRoll = "assets/b6.png"
```

```
     die1 = (secrets.randbelow(5)+1)
             if ev.type == dust_clear_event:
                 state = checkWinner(die1, die2)
                 if state == 0:
                     loseScreen(die1, die2, state, score)
                 elif state == 1:
                     winScreen(die1, die2, state, score)
                 else:
                     drawScreen(die1, state, score)

 def winScreen(die1, die2, state, score):
         cRoll, uRoll = getDice(die1, die2)

 def loseScreen(die1, die2, state, score):
         cRoll, uRoll = getDice(die1, die2)

 def drawScreen(die1, state, score):
         cRoll, uRoll = getDice(die1, die1)

 PS D:\GitHub\high-rollers> 
```

Figure 5: Program slice conducted using the programslicer.py file on variable "die1".

Slicing on variable die2 (User's Roll)

```
PS D:\GitHub\high-rollers> python -u "d:\GitHub\high-rollers\analysis_static\automated\programslicer.py"
Enter the variable to parse for: die2
Program slice on variable:  die2


def checkWinner(die1, die2):
    if (die1 > die2):
        return 0

    elif(die1 < die2):
        return 1

    elif(die1 == die2):
        return 2


def getDice(die1, die2):
    if die2 == 1:
        userRoll = "assets/r1.png"

    elif die2 == 2:
        userRoll = "assets/r2.png"

    elif die2 == 3:
        userRoll = "assets/r3.png"

    elif die2 == 4:
        userRoll = "assets/r4.png"

    elif die2 == 5:
        userRoll = "assets/r5.png"

    elif die2 == 6:
        userRoll = "assets/r6.png"
```

```
     die2 = (secrets.randbelow(5)+1)
             if ev.type == dust_clear_event:
                 state = checkWinner(die1, die2)
                 if state == 0:
                     loseScreen(die1, die2, state, score)
                 elif state == 1:
                     winScreen(die1, die2, state, score)

 def winScreen(die1, die2, state, score):
         cRoll, uRoll = getDice(die1, die2)

 def loseScreen(die1, die2, state, score):
         cRoll, uRoll = getDice(die1, die2)

 PS D:\GitHub\high-rollers> 
```

Figure 6: Program slice conducted using the programslicer.py file on variable "die2".

Manual Program Slice Differences

The manual program slices include additional contextual information and variable statements than those present in the automated slices. This is due to the fact that many related statements do not contain the variable by name directly, but instead another variable required within a related call.
The automated slicer operates on the highrollers.txt file included in the repository, whereas manual slicing can be done on any legible piece of source code. Highrollers.txt was modified in order to use consistent variable names so that the automated slicer would be able to gather as much necessary context as possible, allowing it to actually provide benefit to the user.

It can be argued that manual slicing provides a clearer, more expressive snapshot of the source code that could provide additional benefit to the user, however the detriment to this method is the time required to conduct it. Automated slicing speeds this process up significantly, at the cost of some context.

Dynamic Analysis: Instrumentation

Instrumentation was conducted at the source code level for (almost) all methods. Python includes the datetime module, which makes the calculation of the start and end times of each method call incredibly easy. By inserting time.time() statements throughout the source code at various measurement points, the running time or time spent in each function could be measured. This was done for every function, and the difference from the start of the function to its conclusion is calculated as the `result`. The result is output to a file dynamicLog.txt, with a brief message explaining the function for which the measurement was taken. Dynamic analysis created a new version of the source code containing these statements: hrDynamic.py. Dynamic analysis also required for a revised version of the game driver to be created, hrDDriver.py.

Below is a screenshot of the log generated by conducting a single test execution of the program. Playing the game multiple times will continue to append to the log file as it is ONLY created when the main_menu() function is accessed – something that can only occur during game startup. This screenshot excludes some method timings that are repeatedly called on each screen and often have a value of 0 seconds, such as the text_on_screen() method and the tableGen() method. To include these methods in the timing, simply uncomment the associated file lines in the dynamic analysis version of the program.

Instrumentation Screenshots

Dynamic Analysis Log

dynamicLog.txt - Notepad

File Edit Format View Help

```
Dynamic Analysis began at: March 25, 2022, 21:27:49
Main menu: 0.6595039367675781s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
gameTime function: 0.9368319511413574s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0009765625s
displayScore function: 0.0s
displayScore function: 0.0s
displayScore function: 0.0s
checkWinner function, User Win: 0.0s
updateScore function: 0.0s
gameLogic function if User Win: 2.0213334560394287s
catPath function: 0.0s
getDice function: 0.0s
showDice function: 0.042852163314819336s
displayScore function: 0.0s
catPath function: 0.0s
getDice function: 0.0s
showDice function: 0.008042573928833008s
displayScore function: 0.0s
catPath function: 0.0s
getDice function: 0.0s
showDice function: 0.00897526741027832s
displayScore function: 0.0s
catPath function: 0.0s
getDice function: 0.0s
showDice function: 0.007978200912475586s
displayScore function: 0.0s
catPath function: 0.0s
getDice function: 0.0s
showDice function: 0.007979393005371094s
displayScore function: 0.0s
catPath function: 0.0s
getDice function: 0.0s
showDice function: 0.007998228073120117s
displayScore function: 0.0s
```

. . . repeating middle contents omitted for brevity

```
showDice function: 0.008035421371459961s
displayScore function: 0.0s
catPath function: 0.0s
getDice function: 0.0s
showDice function: 0.008015155792236328s
displayScore function: 0.0s
catPath function: 0.0s
getDice function: 0.0s
showDice function: 0.008020877838134766s
displayScore function: 0.0s
winScreen function on EXIT: 2.6055409908294678s
```

Figure 7: Snapshot of dynamicLog.txt created after dynamic analysis.

Dynamic Analysis Implementation screenshots

```
# Draws text using parameters passed
# (message, font to use, colour to use, surface to draw on, and coordinates to place middle of text)
def text_on_screen(msg, font, colour, surface, x, y):
    f = open("dynamicLog.txt", "a")
    start = time.time()
    textobject = font.render(msg, True, colour) # Creates the text object out of the font
    textrect = textobject.get_rect() # creates a rectangle around the text object
    textrect.midtop = (x,y) # set coordinates of rectangle
    surface.blit(textobject, textrect) # display on surface indicated
    end = time.time()
    result = str(end-start)
    f.write("text_on_screen function: " + result + "s \n")
    f.close()
```

Figure 8: text_on_screen() dynamic analysis inserted timing statements.

```
# Game main menu, with start and exit buttons
def main_menu():
    f = open("dynamicLog.txt", "w") # Creates dynamic log for OVERWRITE for current run
    f.truncate() #deletes contents of previous logs in case

    today = datetime.date.today()
    dateForm = today.strftime("%B %d, %Y") # Writes current date in written form
    nowdate = datetime.datetime.now()
    nowString = nowdate.strftime("%H:%M:%S")
    f.write("Dynamic Analysis began at: " + dateForm + ", " + nowString + "\n")
    f.close()

    # Starts timer
    f = open("dynamicLog.txt", "a") # opens dynamic log for APPEND
    start = time.time()  # take the start time measurement after document setup
```

Figure 9.1 : main_menu() starting dynamic analysis inserted timing statements.

```
            if click:
                end = time.time()
                result = str(end-start)
                f.write("Main menu: " + result + "s \n")
                f.close()
                gameTime(0)
        else:
            # Rendering button green
            pygame.draw.rect(screen, gameColours['green'], playButton)
        text_on_screen('play', buttonFont, gameColours['linen'], screen, (width/7)+100, (height/2)+25)

        if quitButton.collidepoint((mx,my)):
            # Rendering button darker red
            pygame.draw.rect(screen, gameColours['dR'], quitButton)
            if click:
                end = time.time()
                result = str(end-start)
                f.write("Main menu to QUIT: " + result + "s \n")
                f.close()
                pygame.quit()
        else:
            # Rendering button light red
            pygame.draw.rect(screen, gameColours['red'], quitButton)
        text_on_screen('quit', buttonFont, gameColours['linen'], screen, ((width/7)+300+width/7), (height/2)+25)


        # Loop to loog for pygame events to exit or move to next screen
        for ev in pygame.event.get():
            if ev.type == pygame.QUIT:
                end = time.time()
                result = str(end-start)
                f.write("Main menu on FORCE QUIT: " + result + "s \n")
                f.close()
                pygame.quit()
                sys.exit()
```

Figure 9.2: Exit (print) statements for main_menu() dynamic analysis.

```
# Generates the rectangle (table) across the bottom of the screen
def tableGen():
    f = open("dynamicLog.txt", "a")
    start = time.time()
    table = pygame.Rect(0, height-250, width, 400)
    pygame.draw.rect(screen, gameColours['brown'], table)
    end = time.time()
    result = str(end-start)
    f.write("tableGen function: " + result + "s \n")
    f.close()
    return True
```

Figure 10: tableGen() dynamic analysis inserted timing statements.

```python
# get appropriate cat path to display in outcome screen
def getCatPath(state):
    f = open("dynamicLog.txt", "a")
    start = time.time()

    if state == 0:
        catPath = "assets/wincat.png"
    elif state == 1:
        catPath = "assets/losecat.png"
    elif state == 2:
        catPath = "assets/draw.png"

    end = time.time()
    result = str(end-start)
    f.write("catPath function: " + result + "s \n")
    f.close()
    return catPath
```

Figure 11: getCatPath dynamic analysis inserted timing statements.

```python
# Updates score based on win/lose/draw
def updateScore(score, status):
    f = open("dynamicLog.txt", "a")
    start = time.time()
    if status == 0:
        score = (score-1)
    elif status == 1:
        score = (score+1)

    end = time.time()
    result = str(end-start)
    f.write("updateScore function: " + result + "s \n")
    f.close()
    return score
```

Figure 12: updateScore() dynamic analysis inserted timing statements.

```python
# Displays the score on the screen
def displayScore(score):
    f = open("dynamicLog.txt", "a")
    start = time.time()

    score = str(score)
    text_on_screen('Score:', scoreFont, gameColours['dY'], screen, (width-90), 30)
    text_on_screen(score, scoreFont, gameColours['dY'], screen, (width-25), 30)

    end = time.time()
    result = str(end-start)
    f.write("displayScore function: " + result + "s \n")
    f.close()
```

Figure 13: displayScore() dynamic analysis inserted timing statements.

```
# Begins the "lets roll" screen of game, with button to start
def gameTime(score):
    f = open("dynamicLog.txt", "a")
    start = time.time()
```

Figure 14.1: gameTime() dynamic analysis inserted timing statements to start tracking.

.

```
        # Hover on roll button
        if rollButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dR'], rollButton)
            if click:
                end = time.time()
                result = str(end-start)
                f.write("gameTime function: " + result + "s \n")
                f.close()
                gameLogic(score)
        else:
            pygame.draw.rect(screen, gameColours['red'], rollButton)

        text_on_screen('ROLL', buttonFont, gameColours['linen'], screen, (width/3)+115, (height-90))

        for ev in pygame.event.get():
            if ev.type == pygame.QUIT:
                end = time.time()
                result = str(end-start)
                f.write("gameTime function on FORCE QUIT: " + result + "s \n")
                f.close()
                pygame.quit()
                sys.exit()
```

Figure 14.2: gameTime() dynamic analysis inserted timing to print results and stop timer.

```
# checks winner based on dice inputs
def checkWinner(roll1, roll2):
    f = open("dynamicLog.txt", "a")
    start = time.time()

    if (roll1 > roll2): # Computer wins
        end = time.time()
        result = str(end-start)
        f.write("checkWinner function, Computer Win: " + result + "s \n")
        f.close()
        return 0

    elif(roll1 < roll2):   # User wins
        end = time.time()
        result = str(end-start)
        f.write("checkWinner function, User Win: " + result + "s \n")
        f.close()
        return 1

    elif(roll1 == roll2): # Draw
        end = time.time()
        result = str(end-start)
        f.write("checkWinner function, Draw: " + result + "s \n")
        f.close()
        return 2
```

Figure 15: checkWinner() dynamic analysis inserted timing statements.

```python
# Obtains image path for computer and user rolled dice
def getDice(die1, die2):
    f = open("dynamicLog.txt", "a")
    start = time.time()

    if die1 == 1:
        compRoll = "assets/b1.png"
    elif die1 == 2:
        compRoll = "assets/b2.png"
    elif die1 == 3:
        compRoll = "assets/b3.png"
    elif die1 == 4:
        compRoll = "assets/b4.png"
    elif die1 == 5:
        compRoll = "assets/b5.png"
    elif die1 == 6:
        compRoll = "assets/b6.png"


    if die2 == 1:
        userRoll = "assets/r1.png"
    elif die2 == 2:
        userRoll = "assets/r2.png"
    elif die2 == 3:
        userRoll = "assets/r3.png"
    elif die2 == 4:
        userRoll = "assets/r4.png"
    elif die2 == 5:
        userRoll = "assets/r5.png"
    elif die2 == 6:
        userRoll = "assets/r6.png"

    end = time.time()
    result = str(end-start)
    f.write("getDice function: " + result + "s \n")
    f.close()
    return (compRoll, userRoll)
```

Figure 16: getDice() dynamic analysis inserted timing statements.

```python
# Displays dice based on img path
def showDice(compRoll, userRoll):
    f = open("dynamicLog.txt", "a")
    start = time.time()

    # resize dice
    userRollR = pygame.transform.scale((pygame.image.load(userRoll).convert_alpha()), (80, 80))
    compRollR = pygame.transform.scale((pygame.image.load(compRoll).convert_alpha()), (80, 80))

    # display user and computer dice
    screen.blit(userRollR, ((width/2)+45, (height/2)+200))
    screen.blit(compRollR, ((width/4)+105, (height/2)+150))
    pygame.display.update()

    end = time.time()
    result = str(end-start)
    f.write("showDice function: " + result + "s \n")
    f.close()
```

Figure 17: showDice() dynamic analysis inserted timing statements.

```python
# generate rolls, next screen navigation
def gameLogic(score):
    f = open("dynamicLog.txt", "a")
    start = time.time()
```

Figure 18.1: gameLogic() dynamic analysis inserted timing statements to start the timer.

```python
        for ev in pygame.event.get():
            if ev.type == pygame.QUIT:
                end = time.time()
                result = str(end-start)
                f.write("gameLogic function on FORCE QUIT: " + result + "s \n")
                f.close()
                pygame.quit()
                sys.exit()

            if ev.type == dust_clear_event:
                # Once dust can clear, navigate to next screen for victory/loss/draw
                winner = checkWinner(die1, die2)
                score = updateScore(score, winner)
                if winner == 0:
                    end = time.time()
                    result = str(end-start)
                    f.write("gameLogic function if Computer Win: " + result + "s \n")
                    f.close()
                    loseScreen(die1, die2, winner, score)
                elif winner == 1:
                    end = time.time()
                    result = str(end-start)
                    f.write("gameLogic function if User Win: " + result + "s \n")
                    f.close()
                    winScreen(die1, die2, winner, score)
                else:
                    end = time.time()
                    f.write("gameLogic function if Draw: " + result + "s \n")
                    f.close()
                    drawScreen(die1, winner, score)
```

Figure 18.2: gameLogic() dynamic analysis inserted timing to print results and stop timer.

```python
# user won
def winScreen(die1, die2, num, score):
    f = open("dynamicLog.txt", "a")
    start = time.time()
```

Figure 19.1: winScreen() dynamic analysis inserted timing statements to start the timer.

```python
        # hover effects (collision)
        if againButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dG'], againButton)
            if click:
                end = time.time()
                result = str(end-start)
                f.write("winScreen function on PLAY AGAIN: " + result + "s \n")
                f.close()
                gameTime(score)
        else:
            pygame.draw.rect(screen, gameColours['green'], againButton)
        text_on_screen('replay', buttonFont, gameColours['linen'], screen, (width/7)+115, (height-75))

        if quitButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dR'], quitButton)
            if click:
                end = time.time()
                result = str(end-start)
                f.write("winScreen function on EXIT: " + result + "s \n")
                f.close()
                pygame.quit()
                sys.exit()
        else:
            pygame.draw.rect(screen, gameColours['red'], quitButton)
        text_on_screen('quit', buttonFont, gameColours['linen'], screen, (width/7)+415, (height-75))

        # event loop looking for click or escape
        for ev in pygame.event.get():
            if ev.type == pygame.QUIT:
                end = time.time()
                result = str(end-start)
                f.write("winScreen function on FORCE QUIT: " + result + "s \n")
                f.close()
                pygame.quit()
                sys.exit()
```

Figure 19.2: winScreen() dynamic analysis inserted timing to print results and stop timer.

```python
# screen for when user loses
def loseScreen(die1, die2, num, score):
    f = open("dynamicLog.txt", "a")
    start = time.time()
```

Figure 20.1: loseScreen() dynamic analysis inserted timing statements to start the timer.

```python
            # hover collision
            if againButton.collidepoint((mx, my)):
                pygame.draw.rect(screen, gameColours['dG'], againButton)
                if click:
                    end = time.time()
                    result = str(end-start)
                    f.write("loseScreen function on PLAY AGAIN: " + result + "s \n")
                    f.close()
                    gameTime(score)
            else:
                pygame.draw.rect(screen, gameColours['green'], againButton)
            text_on_screen('replay', buttonFont, gameColours['linen'], screen, (width/7)+115, (height-75))

            if quitButton.collidepoint((mx, my)):
                pygame.draw.rect(screen, gameColours['dR'], quitButton)
                if click:
                    end = time.time()
                    result = str(end-start)
                    f.write("loseScreen function on EXIT: " + result + "s \n")
                    f.close()
                    pygame.quit()
                    sys.exit()
            else:
                pygame.draw.rect(screen, gameColours['red'], quitButton)
            text_on_screen('quit', buttonFont, gameColours['linen'], screen, (width/7)+415, (height-75))

            # event loop looking for click or escape
            for ev in pygame.event.get():
                if ev.type == pygame.QUIT:
                    end = time.time()
                    result = str(end-start)
                    f.write("loseScreen function on FORCE QUIT: " + result + "s \n")
                    f.close()
```

Figure 20.2: loseScreen() dynamic analysis inserted timing to print results and stop timer.

```
# screen for when computer and user dice are equal
def drawScreen(die1, num, score):
    f = open("dynamicLog.txt", "a")
    start = time.time()
```

Figure 21.1: drawScreen() dynamic analysis inserted timing statements to start the timer.

```
        if againButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dG'], againButton)
            if click:
                end = time.time()
                result = str(end-start)
                f.write("drawScreen function on PLAY AGAIN: " + result + "s \n")
                f.close()
                gameTime(score)
        else:
            pygame.draw.rect(screen, gameColours['green'], againButton)
        text_on_screen('replay', buttonFont, gameColours['linen'], screen, (width/7)+115, (height-75))

        if quitButton.collidepoint((mx, my)):
            pygame.draw.rect(screen, gameColours['dR'], quitButton)
            if click:
                end = time.time()
                result = str(end-start)
                f.write("drawScreen function on EXIT: " + result + "s \n")
                f.close()
                pygame.quit()
                sys.exit()
        else:
            pygame.draw.rect(screen, gameColours['red'], quitButton)
        text_on_screen('quit', buttonFont, gameColours['linen'], screen, (width/7)+415, (height-75))


        for ev in pygame.event.get():
            if ev.type == pygame.QUIT:
                end = time.time()
                result = str(end-start)
                f.write("drawScreen function on FORCE QUIT: " + result + "s \n")
                f.close()
                pygame.quit()
                sys.exit()
```

Figure 21.2: drawScreen() dynamic analysis inserted timing to print results and stop timer.

Challenges Faced During Implementation

Static analysis proved to be the most challenging, as many of the game's functionality and operational abilities relied on variables that were not being examined within some of the automated program slices. Tailoring the program in order to include additional logical structures proved to be challenging– some function headers could not be included as they had nothing to indicate their relevance to the variable.

The primary lesson learned has already been stated in the ***Manual Program Slice Differences*** section. The manual technique provides a more thorough understanding of the source code from which the slice is derived, however, the automated technique is preferred due to the speed at which it operates. Both techniques proved valuable in debugging and analyzing the source code– particularly centered around bugs relating to the values of individual variables and tracking the incorrect actions between variables.