Software Testing

Lab Assignments

April 07, 2025

Niels Van der Planken
s0191930@ad.ua.ac.be

Thomas Gueutal
s0195095@ad.ua.ac.be

Contents

1 Introduction 1
2 Division of Tasks 1
3 Assignment 06 – Fuzzing 2
3.1 Excercise 1 3
3.1.1 Description 3
3.1.2 How to Run 3
3.2 Exercise 2 4
3.3 Exercise 3 5
3.4 Additional Assignment Context (I) 6
3.5 Exercise 4 6
3.6 Exercise 5 8
3.7 Additional Assignment Context (II) 8
3.8 Exercise 6 8
3.8.1 Defining Valid Inputs 8
3.8.2 Mutation Fuzzer – Implementation 8
3.8.3 Mutation Fuzzer – Results 10
3.9 Exercise 7 (BONUS 10 pts) 12
3.10 Exercise 8 (BONUS 15 pts) 12
3.10.1 What is the limitation of black-box fuzzing? 13
3.10.2 How does SAGE solve this problem? 13
4 Attachments 14
References 17

1 Introduction

This report is a deliverable for one of the testing assignments for the course Software Testing (Master Computer Science 2001WETSWT) at the University of Antwerp in the academic year of 2024-2025.

As per the course description, "*The objective of the lab work of the Software Testing course is to help you learn how you can apply the various testing techniques and test design patterns as discussed during the lectures in practice. You will apply these techniques to a simple Pacman system written in Java. The amount of coding that needs to be done is relatively small: The focus is on testing.*".

This report was written in **Typst**.

2 Division of Tasks

The deliverables are as follows.

1. Code solutions to the tasks, located at **this** Github repo. Note that we add the source code at the end of an assignment as a separate release to the github repo.
2. This report, discussing the solutions and code.

We briefly discuss the division of tasks. We should strive for a half-half split of work between the two team members.

Member	Tasks
Niels	Exercises 1-3, 6 (including report)
Thomas	Exercises 4, 7, 8 (including report)

Table 1: Division of tasks amongst team members.

3 Assignment 06 – Fuzzing

First, we repeat a note made at the start of most assignment files. Then, we repeat the introductory text for fuzzing in the assignment file.

> **Important Note:** Create an archive including your solutions after performing all the exercises. Submit it along with your report. Refer to the "Assignment Guidelines" document for introductory information. Note that this assignment has a **25-point late fee instead of additional exercises**.

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

In this assignment, you will write a black-box fuzzer to fuzz JPacman. Since you will do a black-box fuzzing, you are provided an executable jar file and a sample map file on the blackboard page of this course. In this version of JPacman, we have some new updates:

1. A bug is injected in the code,
2. it loads map files
3. it replays an action sequence.

The jar file can be executed in this way:

```
$ java -jar jpacman-3.0.1.jar sample.map SUUWDE
```

It loads the map `sample.map` and then replays the action sequence `SUUWDE`. In this case, it starts the game (S), moves the players up twice (UU), waits (W), moves down (D), and, finally, it exits the game (E). You can find a complete list of actions in Table 2. In this table p is an instance of the class *Pacman* initiated in the main method. Other characters in the action sequence will be ignored.

Key	Action	Key	Action
E	p.exit()	**Q**	p.quit()
S	p.start()	**W**	Thread.sleep(50)
U	p.up()	**L**	p.left()
D	p.down()	**R**	p.right()

Table 2: The complete action (event) list for *Pacman* instance p.

For simplicity, the jar file uses different exit codes:

- Exit code 0: a normal termination.
- Exit code 1: a crash. The application is not able to handle this input.
- Exit code 10: a rejection. The application is able to handle this invalid input.

3.1 Excercise 1

Question: Write a fuzzer in a language you like (Java, Python, ...). Your fuzzer will do these steps in a loop:

1. generating an input (map) file and a random action sequence.
2. running JPacman with generated inputs
3. checking the exit code of the program.

Your fuzzer can exit the loop after a maximum iteration (MAX) or when a time budget finishes (TIME). For the maps, in this step, your fuzzer only generates random binary files.

Send your fuzzer code.

3.1.1 Description

We use **Python** to write a basic black-box fuzzer for the Pacman `.jar`. See Codeblock 7 or `src/fuzz/random_fuzzer.py` in the project source code for the completely random fuzzer loop implementation.

We make the following assumptions.

1. We must generate an action sequence $\mathcal{A} = a_1 ... a_{L_A}$ of length L_A.
   - We choose $0 \leq L_A \leq K_A$, given some arbitrary maximum sequence length K_A.
   - We choose $\mathcal{A}$ so that it contains only **valid** actions as specified in Table 2.
   - This is a form *behavioral fuzzing*, where we require $\mathcal{A}$ to consist of valid components (actions) that are configured in a possibly unexpected or unintended order. This is in contrast to *robustness fuzzing*, which would allow $\mathcal{A}$ to be a random byte sequence, to intentionally inducing crashes.
2. We must generate a map $\mathcal{M} = m_1 ... m_{L_M}$ that is a byte sequence of length L_M.
   - We choose $0 \leq L_M \leq K_M$, given some arbitrary maximum sequence length K_M.
   - We choose $\mathcal{M}$ so that it contains arbitrary bytes.
   - This satisfies the requirement that "*For the maps, in this step, your fuzzer only generates random binary files.*".

For example, choose $K_A = 4$ and $K_M = 5$. Then $\mathcal{A}$ = "QSEW" and $\mathcal{M}$ = "\x01\x02\x03\x04\x05" are valid fuzzed inputs to the Pacman `.jar`, because each individual character of $\mathcal{A}$ is a valid action, and $\mathcal{M}$ may consist of any (random) bytes at all.

3.1.2 How to Run

Note that the **fuzzer must be run from the directory that it is contained in**. This makes file handling easier, since we can assume the Pacman `.jar` is in that same directory, and we can just read input files and dump output files in the same directory.

Assume the current directory is is the root of the project source code, and the following project file structure holds. See Codeblock 1 for the expected file structure and invocation syntax of the black-box fuzzer.

To specify the quit condition of the fuzzing, we provide two mutually exclusive arguments. Exactly one of these two must be specified.

1. `--TIME x` specifies for how many seconds x the fuzzer may run.
2. `--MAX x` specifies exactly how many fuzzing iterations the fuzzer must perform.

To specify which fuzzer implementation should run, we provide 2+ arguments in the *Fuzzer Type* input argument group. This group will be extended for further exercises. All the flags in this group are mutually exclusive; exactly one fuzzer type must be specified.

1. `--RND` is the flag to run the completely random fuzzer, as defined here in Exercise 1.
1. `--MUT` is the flag to run the mutation fuzzer, as defined here in Exercise 6.

2. `--NOOP` is the flag to run no fuzzer at all; just run the boilerplate around the fuzzing to see that the script works. This is useful because it still checks if the Pacman `.jar` can be found, and validates the inputs you pass to it.

```
# Assume the following file structure.
src/
 └── fuzz/
      ├── manual_fuzzing/  # The manual fuzzing maps, Ex 4.
      ├── constants.py  # Named constants used by the fuzzer
      ├── fuzzer.py     # The entry point for the black-box fuzzer
      ├── jpacman-3.0.1.jar # The Pacman executable .jar
      ├── mutation_fuzzer.py  # The mutation fuzzer loop implementation, Ex 6.
      ├── random_fuzzer.py  # The random fuzzer loop implementation, Ex 1.
      └── sample.map    # The example map provided on blackboard

# You MUST invoke the fuzzer script from its containing directory.
# Assume we start in the project root.
$ cd src/fuzz/
src/fuzz$ python3 fuzzer.py --NOOP --MAX 1000  # Dry run, call no fuzzer.
src/fuzz$ python3 fuzzer.py --RND --MAX 1000   # completely random fuzzer
src/fuzz$ python3 fuzzer.py --MUT --MAX 1000   # mutation random fuzzer

# When in doubt, run the help command.
$ cd src/fuzz/
src/fuzz$ python3 fuzzer.py --help
```

Codeblock 1: The expected project file structure, and invocation syntax for the black-box fuzzer.

3.2 Exercise 2

Question: Run your fuzzer, which generates pure random inputs (MAX=1000). What do you see?

We perform the following commands to run the completely random fuzzer for 1000 iterations.

```
$ cd src/fuzz/
src/fuzz$ python3 fuzzer.py --RND --MAX 1000
```

We observe that all completely randomly fuzzed maps $\mathcal{M}$ are malformed in some way. Consequently the executable's behavior is not probed very deeply. We list a summary of the fuzzer output, see Codeblock 3.

Every single one of the invocations of the Pacman `.jar` returned **exit code 10**, which implies that the `.jar` never crashed unexpectedly. We observe three distinct error messages.

- There were 983 occurrences of `**** Error reading file by readAllLines`.
- There were 8 occurrences of `**** No lines`. This error occurred when the input map $\mathcal{M}$ was the empty byte string, meaning $\mathcal{M}$ was of length $L_M = 0$.
- There were 9 occurrences of `**** Unknown character`. The cause of this error is uncertain, but we suspect it is still related to the choice of map $\mathcal{M}$, because we only the probability of generating a valid map with a random byte string is very small.

We make an **additional observation**. If we manually fuzz JPacman with the valid `sample.map` map and an action string consisting a mix of valid and invalid actions w.r.t. Table 2, then the program indeed seems to ignore the invalid actions as intended. **But**, if the map is valid, and the action string does **not** contain an exit action 'E', then the program (GUI) never terminates. For the related commands, see Codeblock 2.

```
# Go to where the Pacman .jar and sample map are located.
$ cd src/fuzz/

# Invalid actions '1' are ignored; the program does not terminate without an
exit action 'E'!
src/fuzz$ java -jar jpacman-3.0.1.jar sample.map 1111

# There is a start action 'S' AND exit action 'E'; invalid actions '1' are
ignored and the program terminates successfully.
src/fuzz$ java -jar jpacman-3.0.1.jar sample.map 11S11E
```

Codeblock 2: Manual fuzzing w.r.t. a valid map $\mathcal{M}$ and an arbitrary action string $\mathcal{A}$.

```
src/fuzz$ python3 fuzzer.py --RND --MAX 1000
#= Fuzzing config =#
MAX     1000
TIME    None
RND     True
NOOP    False
#==================#

Start fuzzing at: Thu Apr  3 12:07:38 2025
**** Error reading file by readAllLines  # 983 / 1000 occurrences
...
**** No lines                            # 8 / 1000 occurrences
...
**** Unknown character                   # 9 / 1000 occurrences
...
Stop fuzzing at:  Thu Apr  3 12:08:32 2025
```

Codeblock 3: A summary of the completely random fuzzer output.

3.3 Exercise 3

Question: Do you think a completely random fuzzing is efficient? Why?

Depending on the complexity of the input space, completely random fuzzing may be efficient.

Fuzzing with completely random inputs is **much simpler to implement** than intelligently choosing the fuzzed inputs. This simplicity makes implementing the fuzzer fairly trivial. For **very simple input domains**, for example when a program only takes in three whole number arguments $x, y, z \in \mathbb{Z}$, this fuzzing approach may be efficient, and sufficient to uncover faults.

For the **Pacman program**, the input space is all possible map $\mathcal{M}$ files times all possible action sequence $\mathcal{A}$ configurations. This **input space is too complex for completely random fuzzing to efficiently uncover useful results**. Notably, if the completely randomly fuzzed map $\mathcal{M}$ is not a valid map file, then we can not penetrate deeply into the program's logic, no matter what action sequence $\mathcal{A}$ was generated. Thus, both map $\mathcal{M}$ and actions $\mathcal{A}$ must coincidentally be valid and uncover a fault. This is unlikely given the large input space.

Relatedly, fuzzing, regardless of how intelligently the inputs are fuzzed, is efficient in generating (combinations of) edge case inputs that the programmer does not think about when intentionally designing a test suite. In that sense, completely random fuzzing is more likely to **generate fuzzed inputs that the programmer would never choose as test data**. So *implementing* completely random fuzzing is arguably an efficient use of time w.r.t. generating edge-case, fault-inducing input data.

Finally, completely random fuzzing is a useful tool to **map out the initial pseudocode of an executable when performing black-box fuzzing**, as will be done in the following exercises. It provides the executable with mostly malformed inputs, which allows the tester to get a feel for what the executable considers malformed vs well-formed inputs. This is another efficient use of time for the programmer.

3.4 Additional Assignment Context (I)

Let us do some manual fuzzing and try to guess what is going on behind the scene based on the received errors. We can use this information to make the fuzzer more efficient.

As an example, when we use a binary file, we see the error "*Error reading file by readAllLines*". We can infer the program rejects all non-text files by this error. Then we provide a text file with multiple lines. We see the error changes to "*Widths mismatch*". Based on these errors, we can guess some part of the code in the system under test and build pseudocode like this (pay attention to the order of the checks):

```
if(file is not a text file)
    reject(Error reading file by readAllLines);
if(line lengths are not identical)
    reject(Widths mismatch);
```

3.5 Exercise 4

Question: Continue manual fuzzing. Try corner cases like an empty file, empty lines, special characters, etc. Find as many errors as you can and their order. Describe the activities you performed and write a pseudocode.

Note that if the map is valid, and the action sequence does not contain an exit action, then the program never terminates. Call the following commands to verify this behavior.

```
# Ignore invalid actions '1'.
# Actions sequence does not contain EXIT action 'E'!
# => program does not terminate.
java -jar jpacman-3.0.1.jar sample.map 11S11

# Ignore invalid actions '1'.
# Actions sequence does contain EXIT action 'E'.
# => program terminates
java -jar jpacman-3.0.1.jar sample.map 11S1E1
```

To continue manual fuzzing, different `.map` files were created. We created the files, because otherwise, the files would get rejected as mentioned in section 3.4. The files were bundled in the folder `src/fuzz/manual_fuzzing` which means that if you want to run the files for this exercise, you should use the following command:

```
# Manual fuzzing, change NAME by the filename
java -jar jpacman-3.0.1.jar ./manual_fuzzing/NAME.map SRRDRRDE

# check how the previous line was handled
echo $?
```

For each map, we show an example in Figure 1. These examples do not necessarily mimic the actual map files, but simply illustrate how each map should look like. We also annotate each map section with its corresponding example, e.g. map file i corresponds to map example ($\mathcal{M}_i$).

Empty File ($\mathcal{M}_1$)
The first thing we tried was giving an empty file `empty.map` to the parser. The program handled this by returning the output `**** No lines` and it was finished with exit code 10, meaning that the input was rejected but handled gracefully.

Random File ($\mathcal{M}_2$)
The second file `random.map` contains random symbols. Here the output changed to `**** Unknown Character` and we again had exit code 10. Thus the program knows that the random symbols are an invalid input and therefor rejects it, gracefully.

Different Lengths ($\mathcal{M}_3$)
In this file `dif_lengths.map`, we have different lines of valid input symbols, but every line has a different length, either more or less symbols than the previous one. Again, the program rejects the input with exit code 10 and message `**** Width mismatch`.

Empty End ($\mathcal{M}_4$)
The goal of this file was to have two valid lines and then just an empty line, not finishing the bottom wall for example. The first draft of this file had as first line `WWWWW`, followed by `W000W`, ending with nothing. If this map would load to the game, there wouldn't be a bottom of the playground and the player could "fall off".
In the first draft, no player was set. The program rejected the file because no player was set: `**** No Player set`. Therefore we adapted the file such that a player was present.
After that the player was set, the input got rejected again because no food was present on the map: `**** No Food`.
After we fixed that, something interesting happened: **we got our first bug**! The program exited with code 1, which means it crashed. The crash happened because the index went out of bounds: in our action sequence, the player must go down, but because there is no line defined in the `.map` file that blocks the player to go down (wall), the program tries to make that happen and the player falls of the board. If we would run the exact same file but remove the "D" from the action sequence, the input gets accepted.

Newline characters ($\mathcal{M}_5$)
In this file, we check how the program reacts to `\n`. This again gets handled by the `**** Unknown Character` from before.

Repeated line ($\mathcal{M}_6$)
The goal of this file was to check what would happen if we have multiple players or multiple food on the board. If we have multiple P's in our `.map` file, the file gets rejected with exit code 10 and message: `**** More than one player`. When we adapt such that we have 1 player and multiple food, the file gets accepted, even though we don't have an upper wall line or a down wall line like in the `empty_end.map` file. As long as the action sequence stays within the given board, no crash will occur.

$$\mathcal{M}_1 = \emptyset \qquad \mathcal{M}_2 = \begin{bmatrix} T & 7 \\ 99 & B \end{bmatrix} \qquad \mathcal{M}_3 = \begin{bmatrix} 0 & & \\ 0 & 0 & \\ 0 & 0 & 0 \end{bmatrix}$$

$$\mathcal{M}_4 = \begin{bmatrix} W & W & W & W \\ W & P & F & W \\ & & & \end{bmatrix} \qquad \mathcal{M}_5 = [\backslash\mathrm{n} \;\; \backslash\mathrm{n}] \qquad \mathcal{M}_6 = \begin{bmatrix} 0 & P & F & 0 \\ 0 & P & F & 0 \end{bmatrix}$$

Figure 1: Some arbitrary maps to illustrate different manual fuzzing attempts.

3.6 Exercise 5

Question: Based on the information you have gathered by manual fuzzing, improve your fuzzer. Run your fuzzer to fuzz JPacman (TIME=10 minutes). Send your fuzzer code and the result (only crashes).

Skipped, because we answered the bonus questions instead.

3.7 Additional Assignment Context (II)

Manual fuzzing needs human interaction and it depends on tester experiences. An alternative technique to make the fuzzer more efficient is mutation-based fuzzing. In this technique, the fuzzer will use a valid and well-defined input and mutate some parts of that. Inputs generated by this method are more likely to pass initial checks and test the program deeply.

3.8 Exercise 6

Question: Change your fuzzer to a mutation-based one. Run your fuzzer on JPacman (TIME= 10 minutes). Send your fuzzer code and the result (only crashes).

We reference an article on creating a mutation based fuzzer for inspiration. [1]

3.8.1 Defining Valid Inputs

We must first provide definitions of valid inputs, to generate valid starting points for the mutation process. We intentionally do so again, despite Exercise 1 providing some definitions, because we require more practically-oriented definitions to specify the mutation operations.

A **valid action sequence** $\mathcal{A}$ is defined by the following properties.

- Any character in Table 2 is considered a valid action.
- A valid action sequence $\mathcal{A}$ is of length $0 \leq L_A = |\mathcal{A}|$.
- The final action must be an exist action 'E'. If not, the invoked Pacman program would not terminate and thus not allow the testing to progress.

A **valid map** $\mathcal{M}$ is defined by the following properties.

- Any character in Table 3 is considered a map cell.
- A valid map $\mathcal{M}$ is a well-formed rectangle of valid cells.
  - The width w and height h of the rectangle must be at least 1, i.e. $0 < w, h$.
  - Every column must contain exactly h cells.
  - every row must contain exactly w cells.
  - All map cells in every row and column must be valid.

Cell Character	Description
0	An unoccupied cell.
W	A *Wall* cell.
F	A *Food* cell.
P	A *Player* cell.
M	A *Monster* cell.

Table 3: The list of valid cells when specifying a map $\mathcal{M}$.

3.8.2 Mutation Fuzzer – Implementation

See Codeblock 8 or `src/fuzz/mutation_fuzzer.py` in the project source code for the mutation fuzzer loop implementation.

Much like the random fuzzer implementation, the mutation fuzzer still performs a series of completely independent fuzzing iterations.

Given are an initial map $\mathcal{M}_{w,h} = m_1 ... m_{L_M}$ where $L_M = w \cdot h$ with width $w > 0 \land h > 0$ or $w = 0 \land h = 0$, and an initial action sequence $\mathcal{A} = a_1 ... a_{L_A}$ of length $L_A \geq 0$. We also call these the seed values, since they are the starting point for mutations.

We define a set of basic mutation operations, see **Table 6**, which the mutation fuzzer makes use of. We add very basic unit tests in `src/fuzz/test.py` to check that these operations at least produce values of the expected dimensions (length). All of these mutations follow three principles: "add", "delete" or "replace". We simply create specialized variants for $\mathcal{A}$ and $\mathcal{M}$.

Mutation fuzzing always determines some seed values before fuzzing starts. The seed values represent the template onto which mutations will be applied during the fuzzing, to produce the fuzzed inputs. Starting from valid values drastically increases the chance of generating valid, well-formed fuzz inputs.

Note that we randomize the number of mutations that are applied to the seed values during each distinct fuzzing iteration. In other words, given a minimum number of mutations N_{min} and a maximum number of mutations N_{max}, during the i-th fuzzing iteration we uniformly sample two numbers in the discrete range of allowed mutations counts: $n_A, n_M \in [N_{\text{min}}, N_{\text{max}}]$ where $1 \leq N_{\text{min}} \leq N_{\text{max}}$. Consequently:

- Exactly n_A $\mathcal{A}$-type mutations are applied to the action sequence seed $\mathcal{A}$ in the i-th fuzz iteration.
- Exactly n_M $\mathcal{M}$-type mutations are applied to the map seed $\mathcal{M}$ in the i-th fuzz iteration.

Stronger still, we uniformly sample which mutation operations to apply in sequence. We sample how many and which $\mathcal{M}$-type mutations to perform in the i-th fuzzing iteration, completely independently from how many and which $\mathcal{A}$-type mutations were sampled. For example, if $N_{\text{min}} = 1$ and $N_{\text{max}} = 10$ so that we sample $n_A = 4$ and $n_M = 3$ independently, then perhaps we uniformly sample the sequence of n_A $\mathcal{A}$-type mutations

$$\begin{aligned} \mathcal{A}' &= \text{sadd}(\mathcal{A}) \\ \mathcal{A}'' &= \text{srpl}(\mathcal{A}') \\ \mathcal{A}''' &= \text{sadd}(\mathcal{A}'') \\ \mathcal{A}'''' &= \text{sdel}(\mathcal{A}''') \end{aligned} \qquad \begin{aligned} \mathcal{A} &= \text{UUDE} \\ \mathcal{A}' &= \text{SUUDE} \\ \mathcal{A}'' &= \text{SURDE} \\ \mathcal{A}''' &= \text{SURDES} \\ \mathcal{A}'''' &= \text{SRDES} \end{aligned}$$

and the sequence of n_M $\mathcal{M}$-type mutations.

$$\begin{aligned} \mathcal{M}' &= \text{cadd}(\mathcal{M}) \\ \mathcal{M}'' &= \text{mrpl}(\mathcal{M}') \\ \mathcal{M}''' &= \text{rdel}(\mathcal{M}'') \end{aligned} \qquad \mathcal{M} = \begin{bmatrix} W \\ P \\ F \end{bmatrix} \; \mathcal{M}' = \begin{bmatrix} W & 0 \\ P & \text{F} \\ F & \text{M} \end{bmatrix} \; \mathcal{M}'' = \begin{bmatrix} W & \text{W} \\ P & F \\ F & M \end{bmatrix} \; \mathcal{M}''' = \begin{bmatrix} W & W \\ F & M \end{bmatrix}$$

Here $\mathcal{A}$ and $\mathcal{M}$ were chosen arbitrarily, as seed values.

Operation	Description
	(3) $\mathcal{A}$-type mutation operations
$sadd(\mathcal{A})$	Adds one valid action at any random position of $\mathcal{A}$.
$sdel(\mathcal{A})$	Deletes one action at any random position of $\mathcal{A}$ iff. $L_A > 0$. Else simply return $\mathcal{A}$ unmodified.
$srpl(\mathcal{A})$	Replaces one action at any random position of $\mathcal{A}$ by any random, valid action iff. $L_A > 0$. Else simply return $\mathcal{A}$ unmodified.
	(5) $\mathcal{M}$ -type mutation operations
$cadd(\mathcal{M})$	Adds one column of valid cells at any random column position of $\mathcal{M}$ iff. $L_M > 0$. Else simply return $\mathcal{M}$ unmodified.
$radd(\mathcal{M})$	Adds one row of valid cells at any random row position of $\mathcal{M}$ iff. $L_M > 0$. Else simply return $\mathcal{M}$ unmodified.
$cdel(\mathcal{M})$	Deletes one column of cells at any random column position of $\mathcal{M}$ iff. $L_M > 0$. Else simply return $\mathcal{M}$ unmodified.
$rdel(\mathcal{M})$	Deletes one row of cells at any random row position of $\mathcal{M}$ iff. $L_M > 0$. Else simply return $\mathcal{M}$ unmodified.
$mrpl(\mathcal{M})$	Replace one cell at any random row plus column position of $\mathcal{M}$ iff. $L_M > 0$. Else simply return $\mathcal{M}$ unmodified.

Table 6: The mutation operations, by type ($\mathcal{A}$ or $\mathcal{M}$).

3.8.3 Mutation Fuzzer – Results

We now run the mutation fuzzer. In Figure 2 we specify the used seed values for action sequence $\mathcal{A}$ and map $\mathcal{M}$. Note that the $\mathcal{M}$ map seed is exactly the sample map provided to us for this assignment. The action sequnce was chosen manually, to be long enough to perform several consecutive mutations on and still retain the exit action 'E' as final action in the sequence with reasonable frequency, so that the sequence is valid.

We choose $N_{\text{min}} = 1$ minimum mutations and $N_{\text{max}} = 10$ maximum mutations.

In other words, every single iteration of the fuzzer will use the seed values for $\mathcal{A}$ and $\mathcal{M}$, see Figure 2, as a starting point, and then apply mutations to those starting values to obtain the fuzzed values for that independent iteration. Then we invoke the Pacman `.jar` with the fuzzed inputs $\mathcal{A}^{(n_A)}$ and $\mathcal{M}^{(n_M)}$.

$$\mathcal{A} = \text{SUELDUWRUESLUWDE}$$

$$\mathcal{M} = \begin{bmatrix} W & W & W & W & W & W & W \\ W & 0 & 0 & 0 & 0 & 0 & W \\ W & 0 & 0 & M & 0 & 0 & W \\ W & 0 & P & 0 & 0 & 0 & W \\ W & 0 & 0 & 0 & F & 0 & W \\ W & W & W & W & W & W & W \end{bmatrix}$$

Figure 2: The seed actions $\mathcal{A}$ and map $\mathcal{M}$ values.

To validate that the seed values are well-formed, we can pass them to the jar. See Codeblock 4 and Figure 3 to see the relevant command, and that it executes without errors.

Finally, we invoke the mutation fuzzer to run for 10 minutes (600s) with the following command.

```
src/fuzz$ fuzzer.py --MUT --TIME 600
```

We encounter two distinct exceptions throw by the Pacman `.jar`. We list them in Codeblock 5. Additionally, we give an example for each exception of which inputs trigger it, see Table 7. The fuzz values $\mathcal{A}_1^{(n_A)}$ and $\mathcal{M}_1^{(n_M)}$ induce the `IllegalArgumentException` and $\mathcal{A}_2^{(n_A)}$ and $\mathcal{M}_2^{(n_M)}$ induce the `ArrayIndexOutOfBoundsException`. To make this concrete, we show that manually invoking JPacman on the CLI with the specified, fuzzed inputs produces the errors as shown, see Figure 4.

```
src/fuzz$ java -jar jpacman-3.0.1.jar sample.map SUELDUWRUESLUWDE
```

Codeblock 4: The command to manually run the JPacman `.jar` with the seed values.

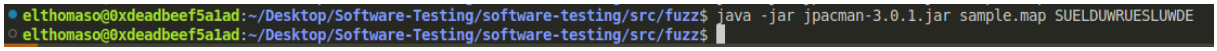

Figure 3: Actually run the JPacman `.jar` with the seed values, see Codeblock 4.

```
# Exception (1)
java.lang.IllegalArgumentException: bound must be positive
        at java.base/java.util.Random.nextInt(Random.java:388)
        ...

# Exception (2)
java.lang.ArrayIndexOutOfBoundsException: Index -1 out of bounds for length 6
        at jpacman.model.Board.getCell(Board.java:89)
        at jpacman.model.Cell.cellAtOffset(Cell.java:165)
        at jpacman.model.Game.movePlayer(Game.java:284)
        at jpacman.model.Engine.movePlayer(Engine.java:203)
        at jpacman.controller.Pacman.left(Pacman.java:173)
        at jpacman.controller.Pacman.mainFuzzing(Pacman.java:260)
        at jpacman.controller.Pacman.main(Pacman.java:205)
```

Codeblock 5: The two distinct exceptions thrown by JPacman.

$$\mathcal{A}_1^{(n_A)} = \text{LUELDUWRUESLUWDEE} \qquad \mathcal{A}_2^{(n_A)} = \text{SUESDUWUESLLUWDE}$$

$$\mathcal{M}_1^{(n_M)} = \begin{bmatrix} W & W & W & W & W \\ W & 0 & P & W & W \\ W & 0 & 0 & 0 & W \\ W & 0 & 0 & 0 & W \\ W & 0 & 0 & 0 & W \\ W & 0 & F & 0 & W \\ W & W & W & W & W \end{bmatrix} \qquad \mathcal{M}_2^{(n_M)} = \begin{bmatrix} W & W & W & W & W & W \\ P & F & F & 0 & W & 0 \\ W & 0 & 0 & 0 & 0 & W \\ W & 0 & M & 0 & 0 & W \\ W & 0 & 0 & 0 & 0 & W \\ W & 0 & 0 & F & 0 & W \\ W & W & W & W & W & W \end{bmatrix}$$

Table 7: The fuzzed inputs that induce the exceptions in Codeblock 5. The fuzz values $\mathcal{A}_1^{(n_A)}$ and $\mathcal{M}_1^{(n_M)}$ induce the `IllegalArgumentException` and $\mathcal{A}_2^{(n_A)}$ and $\mathcal{M}_2^{(n_M)}$ induce the `ArrayIndexOutOfBoundsException`.

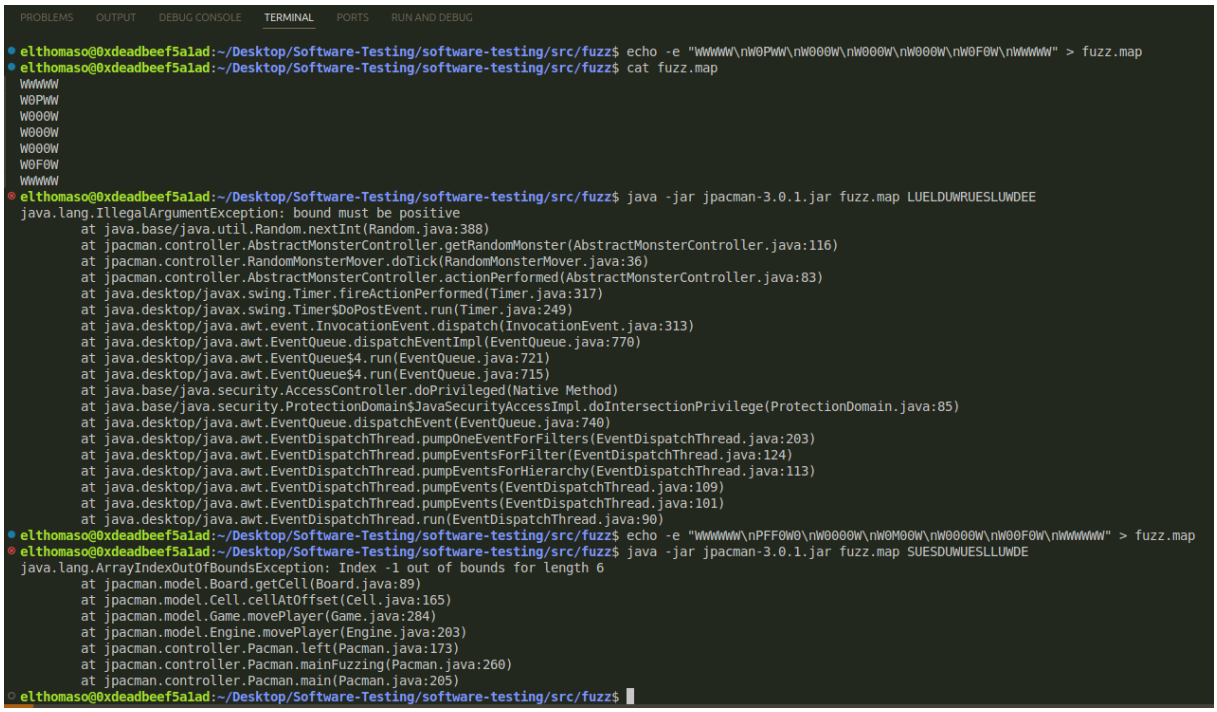

Figure 4: Manually invoking JPacman with the fuzzed inputs of Table 7 to induce the errors in Codeblock 5.

3.9 Exercise 7 (BONUS 10 pts)

Question: Watch this video _https://www.youtube.com/watch?v=1S0aBV-Waeo about the buffer overflow attack in C. How can fuzzing help a security tester to find buffer overflow vulnerabilities?

Fuzzing is a dynamic software testing technique that helps security testers identify buffer overflow vulnerabilities by automatically injecting a wide range of unexpected or malformed inputs into a program and monitoring its behavior for anomalies such as crashes or memory errors. This process is particularly effective in uncovering buffer overflows, which occur when a program writes more data to a buffer than it can hold, potentially leading to arbitrary code execution or system crashes.

By systematically varying input data, fuzzing can reveal buffer overflows that might be missed by manual code reviews or static analysis. For example, fuzzing tools can automate the process of delivering corrupted input and closely watch for unexpected responses from the application, making it easier to detect such vulnerabilities.

Moreover, combining fuzzing with sanitizers such as *AddressSanitizer* enhances the detection of memory-related errors. Sanitizers instrument the code to catch issues like buffer overflows at runtime, providing real-time feedback that allows the fuzzer to refine its inputs and uncover even more vulnerabilities.

In summary, fuzzing assists security testers in identifying buffer overflow vulnerabilities by automating the input testing process, increasing code coverage, and leveraging runtime analysis tools to detect and diagnose memory-related issues effectively.

Extra sources are [2], [3], [4], [5].

3.10 Exercise 8 (BONUS 15 pts)

Question: Read the article about SAGE (_https://patricegodefroid.github.io/public_psfiles/cacm2012.pdf), a white-box fuzzing technique, and answer these questions:

3.10.1 What is the limitation of black-box fuzzing?

Black-box fuzzing, while effective in certain scenarios, has notable limitations. Specifically, it often results in lower code coverage and may miss critical security bugs. For example, consider the following code snippet:

```
int foo(int x) {
// x is an input
  int y = x + 3;
  if (y == 13) abort(); // error
  return 0;
}
```

In this function, the `abort()` statement is executed only when `x == 10`. Given that x is a 32-bit integer, randomly selecting values for x has a 1 in 2^32 chance of hitting this specific condition. This low probability illustrates how black-box fuzzing can struggle to exercise certain code paths, thereby missing potential vulnerabilities.

3.10.2 How does SAGE solve this problem?

SAGE (Scalable, Automated, Guided Execution) addresses this limitation through white-box fuzzing techniques. It begins with a well-formed input and dynamically performs symbolic execution on the program under test. During this process, SAGE gathers constraints on inputs from conditional branches encountered along the execution paths. These collected constraints are systematically negated and solved using a constraint solver, generating new inputs that exercise different execution paths in the program. By iteratively applying this method, SAGE can systematically explore many feasible execution paths, significantly increasing code coverage and the likelihood of detecting security vulnerabilities that black-box fuzzing might miss.

4 Attachments

```
"""The constants used in the Pacman black-box fuzzer.

    The following kinds of constants are implemented.
    - The set of valid actions for the Pacman `.jar`.
    - The expected exit codes for the Pacman `.jar`.
    - The path to the Pacman `.jar`
    - Auxiliary file paths.
    - ...
"""

from typing import List

"""The list of valid Pacman actions. Pacman should ignore all other actions."""
ACTIONS: List[str] = [
    "U", # Up               Pacman.up()
    "D", # Down             Pacman.down()
    "L", # Left             Pacman.left()
    "R", # Right            Pacman.right()
    "S", # Start            Pacman.start()
    "E", # Exit             Pacman.exit()
    "Q", # Quit (Halt)      Pacman.quit()
    "W"  # Wait (Sleep)     Thread.sleep(50)
]
ACTIONS_EXIT: str = "E"
assert ACTIONS_EXIT in ACTIONS, "The exit action is not part of the valid actions."

"""The list of valid Pacman cell characters."""
CELLS: List[str] = [
    "0", # An empty cell.
    "W", # A Wall cell.
    "F", # A Food cell.
    "P", # A Player cell.
    "M", # A Monster cell.
]

"""Exit code for a normal termination."""
CODE_OK: int = 0

"""Exit code for a crash. The application is not able to handle this input."""
CODE_ERR: int = 1

"""Exit code for a rejection. The application is able to handle this invalid input."""
CODE_REJ: int = 10

"""The path to the provided Pacman .jar file."""
PATH_PACMAN_JAR: str = "jpacman-3.0.1.jar"

"""The path to the map file generated furing fuzzing."""
PATH_FUZZ_MAP: str = "fuzz.map"

"""The path to the CSV file to dump Pacman error messages into."""
PATH_PACMAN_ERR_CSV: str = "pacman-error.csv"

"""The maximum random length of the fuzzed action sequence."""
LEN_RND_ACTIONS: int = 4

"""The maximum random length of the fuzzed map file contents."""
LEN_RND_MAP: int = 100
```

Codeblock 6: Constants used in the different fuzzers.

```python
"""The completely random fuzzer implementation, for Ex. 1 of the assignment."""
def random_fuzzer(
        curr_progress: Union[int,float],
        stop_progress: Union[int,float],
        update: Callable,
        notdone: Callable):
    with open(cte.PATH_PACMAN_ERR_CSV, 'w') as err_file:
        while notdone(curr_progress, stop_progress):

            # Generate (FUZZ) random actions.
            # This should be a string of valid ACTIONS.
            # Randomize the action sequence length to expose bugs related to:
            # zero, one, many, etc. actions.
            fuzz_actions_len: int = random.randint(0, cte.LEN_RND_ACTIONS)
            fuzz_actions: str = "".join(random.choices(cte.ACTIONS,
k=fuzz_actions_len))

            # Use a file context to close the file even in case of exceptions.
            fuzz_map: bytes = b''
            with open(cte.PATH_FUZZ_MAP, 'wb') as map_file:
                # Generate (FUZZ) random map contents.
                # This should be a random byte string, and maybe a valid map.
                fuzz_map_len: int = random.randint(0, cte.LEN_RND_MAP)
                fuzz_map: bytes = random.randbytes(fuzz_map_len)

                map_file.write(fuzz_map)

            # Use `subprocess.run` as it is the most up-to-date approach.
            result: subprocess.CompletedProcess = subprocess.run([
                "java", "-jar", cte.PATH_PACMAN_JAR, cte.PATH_FUZZ_MAP, fuzz_actions
            ])
            code: int = result.returncode

            # Pacman works fine for the given input.
            # Explicitly do nothing.
            if code == cte.CODE_OK: pass
            # Pacman successfully parsed and rejected this input.
            # Explicitly do nothing.
            elif code == cte.CODE_REJ: pass
            # Pacman failed with an error.
            # Report these Pacman inputs, they indicate a potential bug.
            elif code == cte.CODE_ERR: pass
            else:
                raise RuntimeError(f"Unknown Pacman exit code: {code}")

            # Dump results to file.
            dumptext = f"{code},{fuzz_actions},{fuzz_map}\n"
            err_file.write(dumptext)

            curr_progress = update(curr_progress)
```

Codeblock 7: The completely random fuzzer implementation, for Exercise 1.

```python
"""The mutation fuzzer implementation, for Ex. 6 of the assignment."""
def mutation_fuzzer(
        curr_progress: Union[int,float],
        stop_progress: Union[int,float],
        update: Callable,
        notdone: Callable,
        seed_map: List[str],
        seed_act: str,
        mmin: int,
        mmax: int):

    assert 0 < mmin, "The min nr of mutations is invalid."
    assert 0 < mmax and mmin <= mmax, "The max nr of mutations is invalid."

    with open(cte.PATH_PACMAN_ERR_CSV, 'w') as err_file:
        # Define the mutation operations as a list to sample from randomly.
        action_mutators = [ sadd, sdel, srpl ]
        map_mutators = [ cadd, radd, cdel, rdel, mrpl ]

        while notdone(curr_progress, stop_progress):
            # Mutate the action sequence.
            # Randomize the number of mutations every time.
            mutfuzz_actions: str = seed_act
            for _ in range(random.randint(mmin, mmax)):
                mutfuzz_actions = random.choice(action_mutators)(
                    mutfuzz_actions, cte.ACTIONS
                )

            # The actions sequence must end with an exit action,
            # else JPacman would not terminate!
            if mutfuzz_actions[-1] != cte.ACTIONS_EXIT:
                # Count this as a valid fuzzing attempt!
                curr_progress = update(curr_progress)
                continue

            # Use a file context to close the file even in case of exceptions.
            mutfuzz_map: str = ""
            with open(cte.PATH_FUZZ_MAP, 'w') as map_file:
                # Mutate the map.
                mutfuzz_map_lst: List[str] = seed_map
                for _ in range(random.randint(mmin, mmax)):
                    mutfuzz_map_lst = random.choice(map_mutators)(
                        mutfuzz_map_lst, cte.CELLS
                    )
                mutfuzz_map = "\n".join(mutfuzz_map_lst)

                map_file.write(mutfuzz_map)

            # Use `subprocess.run` as it is the most up-to-date approach.
            result: subprocess.CompletedProcess = subprocess.run([
                "java", "-jar", cte.PATH_PACMAN_JAR, cte.PATH_FUZZ_MAP, mutfuzz_actions
            ])
            code: int = result.returncode

            # Pacman works fine for the given input.
            # Explicitly do nothing.
            if code == cte.CODE_OK: pass
            # Pacman successfully parsed and rejected this input.
            # Explicitly do nothing.
            elif code == cte.CODE_REJ: pass
            # Pacman failed with an error.
            # Report these Pacman inputs, they indicate a potential bug.
            elif code == cte.CODE_ERR: pass
            else:
                raise RuntimeError(f"Unknown Pacman exit code: {code}")

            # Dump results to file.
            dumptext = f"{code},{mutfuzz_actions},\"{mutfuzz_map}\""
            dumptext = dumptext.replace('\n', "\\n")
            dumptext += '\n'
            err_file.write(dumptext)

            curr_progress = update(curr_progress)
```

Codeblock 8: The mutation fuzzer implementation, for Exercise 6.

References

[1] "Mutation-Based Fuzzing." Accessed: Apr. 05, 2025. [Online]. Available: https://www.fuzzingbook.org/html/MutationFuzzer.html

[2] "HDR-Fuzz: Detecting Buffer Overruns using AddressSanitizer Instrumentation and Fuzzing." Accessed: Apr. 06, 2025. [Online]. Available: https://arxiv.org/abs/2104.10466

[3] "Efficient Greybox Fuzzing to Detect Memory Errors." Accessed: Apr. 05, 2025. [Online]. Available: https://arxiv.org/abs/2204.02773

[4] "How to Secure Coding in C and C++ Using Fuzz Testing." Accessed: Apr. 05, 2025. [Online]. Available: https://www.code-intelligence.com/blog/secure-coding-cpp-using-fuzzing

[5] "Wikipedia, Code sanitizer." Accessed: Apr. 05, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Code_sanitizer