Hangman Game – A C-Based Console Application

MD Labib Daiyan Digonto (2521761642), Rahath Imran (2522916042),
Abdullah Al Mamun (2522141), MD Shahed Ahmed (2523286042)
Group 3, Section4

***Abstract*—Abstract—This project details the design, implementation, and verification of a two-player Hangman game in C, created for a command line interface (CLI), as an application of software engineering. The architecture is built on a modular design using functional decomposition and robust state management. One of the main considerations of implementation is secure input, which includes using fgets() to prevent buffer overflow and a multi-tier validation system to validate user input data integrity and reject invalid guess inputs and duplicate guesses. The application has an overall user experience influenced by the implementation of a core game loop that uses finite state machine logic, and features such as an incremental ASCII art representation of current guesses, a dynamic representation of the complete word as it is revealed in real time, and a visual representation of used characters. The deterministic win/loss conditions of the system are based on whether the word guessed is completed by the player or if the player runs out of chances to guess.The completed program fulfills its aims of demonstrating proficiency in the fundamentals of C, including string handling and handling user input/output, and intelligent omission of invalid data through defensive coding, demonstrates and covers systematic validation allowing for the program to be used as a development guide for students looking to develop educational games.**

***Index Terms*—C programming, string validation, hangman, input handling, game logic.**

I. Introduction

A. Background: The Hangman Game as an Educational Paradigm

Hangman is a simple game where one player thinks of a word and the other player tries to guess the word by suggesting letters, with a limited number of guessed letters. Each time the letter guest is wrong it reveals part of a hangman figure. A game is lost if a hangman figure is completed before guessing the word. In computer science education, Hangman is a prominent educational tool for students learning procedural programming in C, because it appears simple, yet it provides students with an opportunity to learn the fundamental concepts of a programming language: working with strings using character arrays; programming logic and iteration such as: `while` and `for` loops) ; conditional statements such as: (`if/else` branching; input/output (I/O) operation; and storing variable states (e.g., remaining chances, tracked letters guessed, the guessed word has been revealed). The reason for the CLI environment for this project, was to be focused on using algorithms, creating data structures, and learning about safe systems without inadvertently being distracted by graphics subsystems.

B. Motivation: Addressing Common Pitfalls in Novice Programming

Novice programmers frequently encounter two critical pitfalls when developing interactive applications: severe security vulnerabilities and unmaintainable monolithic design. The first, and most dangerous, arises from insecure input handling. Functions like `scanf("%s")` are often taught for their simplicity but are notoriously unsafe because they perform no bounds checking. When a program reads user input into a buffer stored on the program's **stack**, an oversized input can lead to a **buffer overflow**. This occurs when the extra data overwrites adjacent memory on the stack, corrupting other local variables or, most critically, the function's **return address**. A malicious actor could exploit this by overwriting the return address to point to their own injected code, enabling arbitrary code execution—one of the most classic and severe security exploits.

The second pitfall is architectural, in that you're writing massive monolithic "spaghetti code" in which all of the logic for the program is just buried in the `main()` function. This structure increases **cognitive load** tremendously, as it makes the code exceptionally difficult for another developer, or even the programmer, to read, debug, and maintain. The reason for the complexity is that the variables all reside in the same scope (the `main()` function), making it very difficult to trace the bug's source, since any part of the program can change any variable. This project is directly concerned with ameliorating these problems, and has integrated best practices into the extremeness of its implementation. We are motivated to not just achieve functional completeness in this project, but to provide a pedagogical exemplar of secure, modular, defensive programming that demonstrates educational projects can have production-grade rigor.

C. Project Objectives and Scope

Primary Objective: To develop a fully functional, secure, and robust two-player Hangman game in C for CLI environments.

Secondary Objectives:

- Implement buffer-overflow-resistant input handling using `fgets()` and `strcspn()`.
- Design a modular architecture with decomposed functions for input, validation, rendering, and game logic.
- Develop a multi-tiered validation system rejecting empty inputs, non-alphabetic characters, and duplicate guesses.
- Create an intuitive CLI user experience featuring dynamic ASCII art, real-time word revelation, and guess tracking.

Scope Boundaries:

- Platform: Exclusively console-based; no graphical libraries (e.g., OpenGL) or external frameworks.
- Word Source: Words are input directly by Player 1; no external dictionary files or APIs.
- Memory Management: Static stack allocation only; dynamic allocation (`malloc`, `free`) is omitted to prevent leaks and simplify resource handling.

D. Report Structure

This report documents the structured development of the project. Section II describes the architecture of the system including data structures and finite-state machines. Section III describes and synthesizes selected aspects of the implementation of the system, but primarily focuses on the secure handling of input and the game logic. Section IV contains the test matrices and analysis of edge-case scenarios to demonstrate the robustness of the implementation as each was tested. Section V describes the engineering issues that occurred and the clever solutions that were devised. Section VI describes possible future development. Finally, in Section VII, we provide some contemplations about the pedagogical issues this project raised.

II. System Design and Architecture

A. Architectural Overview

The overall design of the hangman implementation is a **three-phased sequential model**, a model that enforces modularity, security, and deterministic state transitions. The three-phased model is designed to ensure a deliberate separation of concerns while allowing for a linear, predictable sequence of interactions. The first phase is **Initialization & Setup**, which focuses on safely acquiring the secret word from Player 1. In this first phase, we will establish quite a robust input loop that utilizes multi-tiered validation in ensuring we are successfully two-thirds of the way through the logic checks (buffer limits, character types and empty strings) associated with input capture. When the input loop has accepted the desired character-length of string, we validate the input. Upon validating the input of the first player, we clear the cli (command line interface) by printing 50 blank newlines to scroll any previous text down. This is a simple approach that can cover any platform situation instead of ensuring the word was not seen by subsequent players. This could have been achieved through a number of system commands; e.g., `system("cls")` in Microsoft Windows or `system("clear")` in UNIX-Like systems; however, for portability and security reasons, it is preferable to leave the application free-of any system-specific commands. Since both of the aforementioned commands are system-specific, they can cause portability issues and security issues, which is also why we opted to flood the cli with newlines to allow for an effective 'clear', albeit primitive and old-fashioned, we are guaranteeing the same simple behavior in any standard C-compliant terminal, and thereby, maintaining full-portability!

The system then advances to the second phase known as the **main game loop**. The main game loop is the interactive center of the application. The loop has an invariant condition where play continues while `chances_left > 0` and the word has not been guessed. There is a strict cycle of each loop iteration: render the current state of the game, check whether the win condition has occurred, process a guess from player 2, and update the state with real-time feedback. The **termination** phase occurs when a terminal condition has occurred. A win condition occurs when the entire word is revealed, causing a win message to be displayed. A loss condition occurs when chances are equal to 0. The remaining attempts are no longer available, displaying a game-over message with a reveal of the secret word. The flow of the complete process which models this behavior as a finite-state machine is shown in Fig. 1.

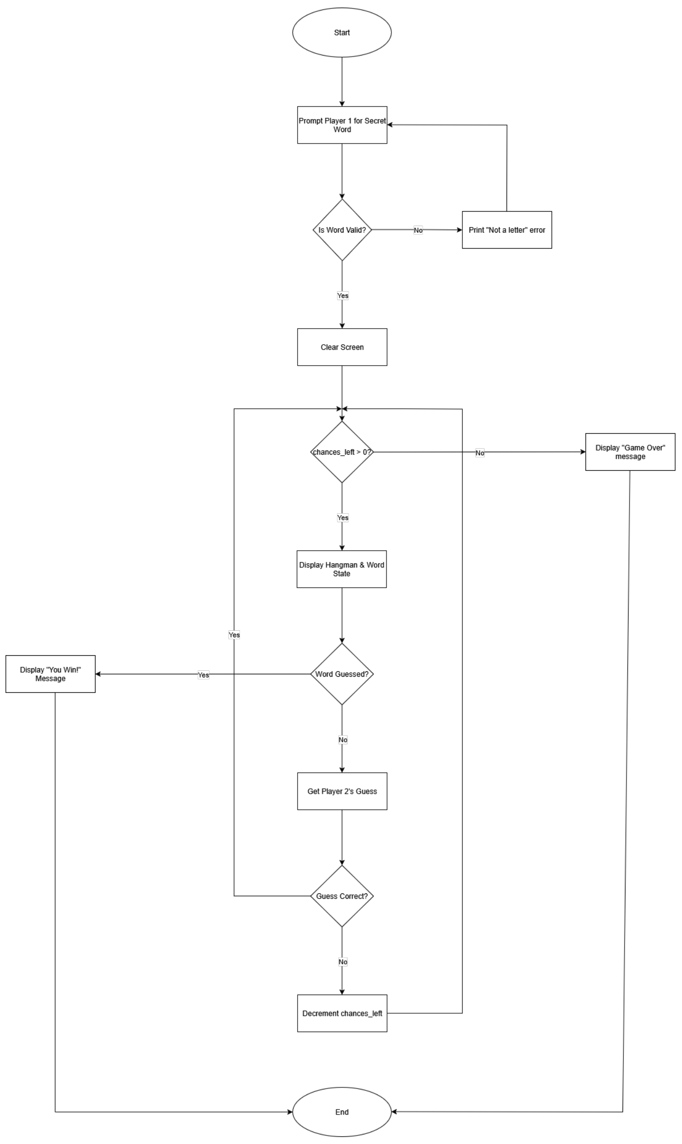

Fig. 1: Finite state machine of the complete Hangman game logic, illustrating the three-phase architecture from initialization to termination.

B. Core Data Structures

The selection of data structures was critical to ensure efficient state management, clarity of intent, and program safety, all while operating within a constrained resource environment.

The secret word itself is contained within `char secret_word[MAX_WORD_LENGTH]`. This is a one-dimensional character array allocated **statically on the stack**. This was done on purpose and was an extremely important design decision after evaluating the pros and cons of static vs. dynamic memory. Although using `malloc()` to allocate memory for the secret word **dynamically on the heap** would provide flexibility in that we could allocate a word of any length, it also adds an unwelcome layer of complexity and hazard that was considered to be inappropriate within the scope of this assignment. The most significant hazards associated with dynamic allocation that we avoided by using static allocation were **memory leaks**, where allocated memory is not explicitly released by calling `free()`, and **dangling pointers**, where allocated memory is freed but the pointer at some later time and is then dereferenced, resulting in erratic behavior and allowing for crashes that often are difficult to debug. Furthermore, dynamic memory allocation introduces a requirement for diligent error checking. `malloc()` can fail and return `NULL` if the system is out of memory. In the context of this assignment where the word length is well-defined, the safety, simplicity, and automatic clearing of memory associated with static stack allocation far outweighed any marginal upside of implementing dynamic memory.

This decision directly affects the program's memory layout and performance. If you allocated all primary data structures on the stack, the memory footprint of the application is allocated entirely at compile-time, and no memory is allocated or deallocated at run-time. The programmers know exactly how much memory is needed at compile-time, and the program will not run beyond its footprint. This stability is especially valuable in embedded systems or environments with limited memory. When `main()` starts, the memory for `secret_word`, `used_letters`, and any other local variables you may have in your program will be allocated automatically. When `main()` returns, the memory will be successfully deallocated automatically. With stack allocation you avoid a whole class of memory management bugs, and it makes your program design simpler.

The `char used_letters[MAX_USED_LETTERS]` array is a linear fixed-size buffer used to track Player 2's progress. Since it is only sized for 27 bytes (26 characters plus a null terminator), it is a key part of the game's logic. It gets searched every time a guess is made to check for duplicate entries and it is used by `print_game_state()` to determine which letters of the `secret_word` to print. The search to check for duplicate entries uses a linear scan which affords a time complexity of O(N) where N is the number of used letters at the time.

While a hash set is a more advanced data structure with O(1) average case characteristics that would be ideal for checking for duplicates. There are a couple of reasons why we used a linear array (i.e. a string). First, N is a small constant term (26) so practically there was no performance difference between doing an O(N) scan vs O(1 lookup). Second, while you can implement a hash set in C, it would involve more code complexity and require us to implement a hashing function along with collision detection and handling. Overall, the linear array approach provided a simpler implementation with less memory usage (for our small scale application), with performance characteristics that were adequate for the application, and I try to follow the engineering principle of not complicating issues unnecessarily.

TABLE II: Summary of Test Case Coverage and Results

Test Category	Key Scenarios Validated	Result
Unit Testing	Input validation, State logic	Pass
Integration Testing	Win/loss paths, User journey	Pass
Boundary Testing	Max-length word, Single-letter word	Pass
Security Testing	Buffer overflow, Stream corruption	Pass

The main loss-condition variable is `int chances_left`, and it is initialized to 6 (which is not chosen out of thin air; it corresponds to the six progressive states of the hangman ASCII art, from the scaffold (`6`) to the man completely rendered (`0`)). The tight coupling between the game state data and its visualization allowing a direct mapping has made the development of the game rather intuitive.

Lastly, the application leveraged the use of booleans, like `bool is_word_valid` and `bool word_is_guessed`, by utilizing the library. The new C standard makes the code easier to read and helps mitigate the cognitive load of if you were to use integer flags (`1` or `0`). This was a capitulation to the intent of producing self-documenting code. The values `true` and `false` clearly show themselves in the if statements and the while loops. You do not have to provide an explanation in comments.

C. Function Decomposition and Modular Design

TABLE I: Function Contracts and Responsibilities

Function	Primary Role	Inputs (In) & Outputs (Out)
`main()`	Orchestrator	Manages high-level flow; calls other functions. No direct return value.
`print_hangman()`	View	**In:** `int chances_left`. **Out:** Renders ASCII art to the console.
`print_game_state()`	View / Controller	**In:** `secret_word`, `used_letters`. **Out:** Displays word state; returns `bool` win status.
`get_player_guess()`	Controller / Validator	**In:** `used_letters`. **Out:** Returns a validated, new, lowercase `char`.

To avoid monolithic code and promote maintainability, the system was decomposed into specialized functional units, each with a clearly defined responsibility, or ”contract.” This modular design is a procedural analogue to the Model-View-Controller (MVC) pattern. Specifically, the **Model** is represented by the core data structures (`secret_word`, `chances_left`, `used_letters`) and the logic that mutates their state. The **View** is implemented by the `print_hangman()` and `print_game_state()` functions, which are responsible for all output to the user’s console. The **Controller** is a combination of the `main()` function, which orchestrates the high-level flow, and `get_player_guess()`, which handles user interaction and input. This separation ensures that changes to the user interface (View) do not necessitate changes to the core game logic (Model), a fundamental tenet of robust software design.

The `main()` function acts as the central **Orchestrator**. It does not perform detailed operations itself; instead, it manages the high-level game flow by calling other functions in the correct sequence and managing the primary state variables. Visualization is handled by two distinct functions. `print_hangman(int)` is a pure **View** function; its sole purpose is to render the ASCII art based on the `chances_left` integer passed to it. The `print_game_state(...)` function is a hybrid view-controller. It renders the word display and used letters but also returns a boolean value indicating if the win condition has been met. This design choice provides a significant efficiency gain by checking for victory during the rendering pass, thus avoiding a separate, redundant loop through the word.

User input is completely abstracted away by `get_player_guess(...)`, which acts as a secure **Validation Gateway**. This function shields the main loop from raw, untrusted data. It enforces rules—requiring a new, alphabetic character—and returns only a sanitized, lowercase character, which the main loop can then process without further validation. This decoupling of responsibilities is paramount: visualization logic is isolated from game logic, and input handling never commingles with state mutation.

```c
while (chances_left > 0)
{
    print_hangman(chances_left);

    if (print_game_state(secret_word, ...))
    {
        printf("\nCongratulations, You Win!\n");
        break;
    }

    char guess = get_player_guess(used_letters,
        ...);

    used_letters[num_used_letters++] = guess;
    if (!strchr(secret_word, guess))
    {
        chances_left--;
    }
}
```

Listing 1: The high-level structure of the main game loop, demonstrating the MVC-like call hierarchy.

This structured approach, as demonstrated in the code snippet above, exemplifies how modular design transforms a complex workflow into a series of maintainable and testable components, resulting in a robust and comprehensible system.

III. Implementation of Key Features

A. *Secure Input Handling: From Theory to Practice*

Secure input handling was a primary focus of the project, and instead of fiendishly simple handling, we created a defensive application that would be robust. When it came time for design choices, one significant choice we made was that we would never utilize `scanf(%s, ...)` to read in the secret word. `scanf` is often used in ”Hello World” examples when demonstrating compiled languages, but does not support bounds checking or white space removal on input. In any event, any use of `scanf` for any sort of string input is extremely unsafe; you’ll leave the application vulnerable to buffer overflow vulnerabilities, which is extremely serious in terms of security failure. When a buffer overflow vulnerability is exploited, the attacker is able to write memories or other data provided by a user, after the end of a character array that was already allocated; the attacker is able to obscure that writing, and counterfeited anything else that was relevant in memory - which may otherwise result in crashing the application, changing variables, or holding a convention of resolved inbound exploit.

To mitigate this threat, the `fgets()` function was employed as the secure alternative. In our implementation, `fgets(secret_word, MAX_WORD_LENGTH, stdin)`, the `MAX_WORD_LENGTH` argument is the crucial security feature, instructing the function to read at most `size - 1` characters and preventing any possibility of a buffer overflow.

However, the security of `fgets()` introduces a minor data sanitation challenge: it frequently stores the trailing newline character (`\n`) in the buffer. If left unhandled, a word like ”hello” would be stored as `"hello\n"`, causing functions like `strlen()` and `strcmp()` to fail unexpectedly.

To address this, an elegant solution using `strcspn()` was implemented. This function finds the precise index of the newline character, allowing it to be replaced with a null terminator (`\0`), which safely and cleanly truncates the string.

This two-step process—secure reading followed by precise sanitization—forms the foundation of the application’s robust input architecture.

B. Multi-Tiered Input Validation System

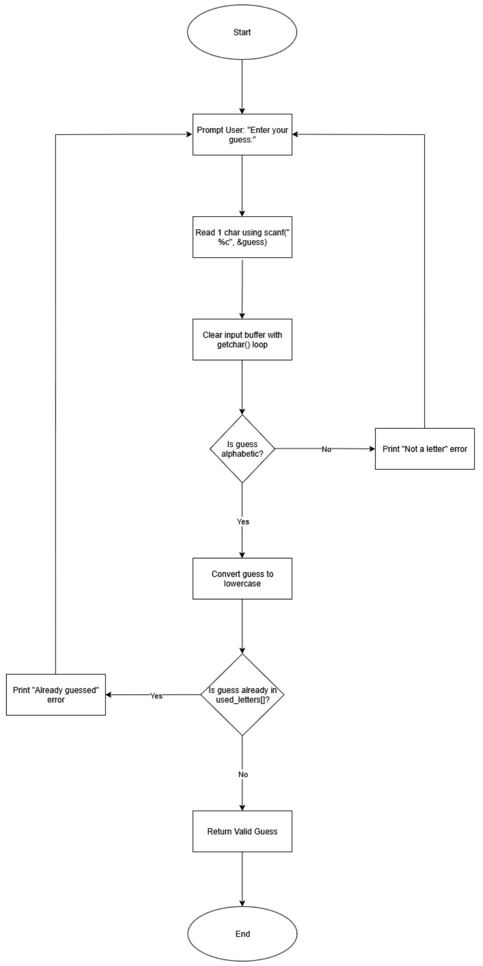
Start
Prompt User: "Enter your guess:"
Read 1 char using scanf(" %c", &guess)
Clear input buffer with getchar() loop
Is guess alphabetic?
No
Print "Not a letter" error
Yes
Convert guess to lowercase
Is guess already in used_letters[]?
Yes
Print "Already guessed" error
No
Return Valid Guess
End

Fig. 2: Detailed logic flow for the 'get player guess()' function

To ensure data integrity and program stability, a multi-tiered validation system was implemented, acting as a filtration gauntlet for all user inputs. This system applies different validation rules tailored to the distinct inputs from Player 1 and Player 2.

For Player 1, the validation of the secret word is handled by a persistent validation engine: a `while(true)` loop that keeps running until a qualified word has been provided. Validation is completed in two levels. **Level 1 (the non-empty check)** - the first and simplest check is `if (strlen(secret) == 0)` - this prevents an empty secret word (unplayable game), and rejects and reprompts the user. **Level 2 (the alphabetic purity check)** - if the string is not empty, the string is then subjected to alphabetic purity check using a `for` loop traversing every character in the string with `isalpha()` function from - enforcing the game rule that the secret word must consist only of letters. The `for` loop was also designed to exit immediately at the first non-alphabetic character by using a `break` statement as an efficiency improvement if the player's letter was non-alphabetic.

Player 2 will validate each guess within the `get()` function, which serves as a validation gateway; however, the validation is even more precise. **Tier 1 (Input Stream Management):** The implementation uses `scanf(" %c", &guess)`. The space in the formatting string is deliberate and critical. When the obvious leading space is specified in the format string, `scanf` will automatically consume any and all whitespace characters and will respect newlines to avoid obvious input stream errors. Also, there is a second, but more subtle, loop `while (getchar() != '\n' && getchar() != EOF)`; the purpose of this is to clear the input buffer. If a user is careless enough to enter a multi-character string like "abc" the guess will be the 'a' but "bc" will be flushed to the input buffer. **Tier 2 (Type Validation):** the type of input will be validated via a simple `if (!isalpha(guess))`. **Tier 3 (Case Validation):** at this point it will standardize the guess by converting it to lowercase (this is all achieved using `guess = tolower(guess))`. Making the whole game case insensitive is a desirable user experience. **Tier 4 (Uniqueness Validation):** finally, there will be a `for`-loop that checks the `used` array and searches to see if the guessed character had been inserted previously. This will enhance player experience by preventing a player from wasting a turn on character previously guessed.

C. The Main Game Loop: The Game's Engine

The core of the interactive gameplay experience is driven by the main game loop, which acts as the application's engine. The loop's structure is defined by the condition `while (chances_left > 0)`, which clearly dictates that the game continues as long as the player has attempts remaining. This loop orchestrates the entire sequence of events that constitutes a single turn.

A single iteration of the loop follows a precise, logical sequence. The iteration **begins** with state display by calling `print_hangman()` and `print_game_state()`. This "display-first" approach is critical for user experience, as it guarantees that the player is always presented with the most current game state—their remaining chances, the revealed parts of the word, and the letters they have already used—before being prompted to make a decision.

Immediately following the state display, the program checks for a win condition. The `if (word_is_guessed)` check, which leverages the boolean value returned from the `print_game_state()` function, provides an efficient and immediate exit path. If the function returns `true`, a victory message is printed, and a `break` statement terminates the main loop. If the game has not been won, the program proceeds to get the next input by calling `get_player_guess()`. This call is guaranteed to return a pre-validated, sanitized character.

Once a valid guess is received, the game state is mutated. The new guess is added to the `used_letters` array, and the `num_used_letters` counter is incremented. Finally, the consequences of the guess are applied. A final `if/else` block checks for the presence of the guess within the secret_word. If the character is found, a positive feedback message is displayed. If it is not found, a negative feedback message is shown, and, crucially, chances_left is decremented.

D. Visualization and User Experience (UX)

The application's user experience is crafted through two key visualization functions that translate the abstract game state into a clear and intuitive console display.

The print() function is responsible for displaying the classic hangman representation. The function is coded as a chain of if/else if/... statements that serves as a visual representation of the current state. Each value of chances_left from 6 to 0 corresponds to a main portion of the ASCII art. When chances_left is 5, a head is displayed; at 4 the body is displayed, and so on. This chaining creates a simple visual representation of the player's remaining lives.

The print() function displays the word itself. The important part of the code is an algorithm that uses a nested loop structure to determine which characters the user has guessed and should therefore be displayed. The **outer loop** iterates through each character of the secret. Each time through the outer loop, a bool match flag is initialized to false. Then, the **inner loop** is executed on the underlying used array. If a character in the secret word matches a character in the used array, the match flag is set to true, and break is executed to end the inner loop. After completing the inner loop, the state of the matchflag determines what value is returned: if true, return the actual character from the secret; otherwise '.', the underscore . This algorithm is case insensitive because of the comparisons tolower(secret[i]) == used[j] allowing for any combination of user's lower case guess to correctly reveal an uppercase letter in the secret word.

IV. Performance and Efficiency Analysis

While the Hangman application is not computationally intensive, design and implementation decisions were made with performance and resource efficiency in mind. This section analyzes the application's memory usage and time complexity.

A. Memory Usage

The application was designed to have a predictable and minimal memory footprint. By using static allocation (on the stack) for all of the application's core data structures, we avoid the overhead, connected fragmentation, and other management issues presented when allocating data on the heap through malloc. Further, all of the memory usage in our application is specified at compile time, and nothing changes during execution. The primary variables in use at any point in time (secret_word, used_letters, integers, booleans, etc.) make up less than 100 bytes of stack usage. This application will be extremely lightweight - it will run in any resource-limited environment (i.e. embedded systems or older hardware) with limited or no heap memory. All of this gives us confidence that the overall program is more stable - there is no entire class of risk from memory management that we could avoid.

B. Time Complexity

The entire runtime of the program is dominated by user input, and since we are dealing with an I/O bound application this means CPU time is essentially zero via the wait state. Nevertheless, the complexity of the core loops was given some thought. The most expensive operation takes place in a nested loop in the print() function displaying the current guess of the word. This algorithm has a time complexity of O(LN), where L is the length of the secret word and N is the number of unique letters guessed thus far. Likewise, the linear search to check for duplicate guesses in get() is O(N). The critical thing is that both L (25 max) and N (26 max) are bounded small constants thus in practice O(1). This guarantees that our application's performance is stable (and doesn't decline as the game progresses) and that we have a responsive and a smooth user experience from the first guess to the last.

V. Testing and Validation

A. Testing Strategy

A detailed and multi-pronged testing strategy was developed to determine the application's functional correctness, security resilience, and behavioral integrity. This strategy combined two main testing approaches: **Unit Testing**, which is a method designed for the specific validation, in isolation, of individual functions against a given set of inputs and edge cases; and **Integration Testing**, which is a method used to establish the end-to-end simulation of full workflows for the games to confirm state transitions and inter-module communications were working correctly. By utilizing both unit testing and integration testing in tandem, errors could be isolated on a component level while knowing there were complete inter-module interactions across the whole system. The first and foremost priorities of the testing phase were: confirm the win/loss logic respectively worked correctly; stress test the input validation toward malicious or incorrect data; stability under edge cases; and complete visual cohesion among all game states.

B. Unit Testing Scenarios

While a formal test harness was not used, the principles of unit testing were applied by creating temporary driver functions to systematically evaluate component behavior in isolation.

The get() function, as the most complex validation gate, was tested heavily. It has been verified that it accepts valid alphabetic characters (for instance, 'a'), then normalizes upper case to lower case (for instance, 'Z' becomes 'z'), and rejects invalid input, like numeric characters ('7') and special characters ('%'), when provided appropriate error messages before prompting the user again. It has been verified that state dependent validation works; for instance, if a player attempts to guess a letter that already exists in the used array, it was properly rejected. And lastly, it has been stress tested to confirm that its input stream handling; it read a multi-character string, ”abc\n”, and properly only processed the first character (’a’) and effectively disposed of the subsequent characters (”bc\n”) from the input stream, ensuring that it did not corrupt subsequent input calls.

The `print_game_state()` function was validated for both its visual output and its logical return value. When called with an empty set of guessed letters, it correctly rendered a series of underscores. When provided with a partial set of correct guesses (e.g., ’t’ and ’e’ for the word ”test”), it accurately displayed a mix of revealed letters and underscores (`"t e _ t"`). Critically, its boolean return logic was confirmed: it consistently returned `false` for any incomplete word and returned `true` only when all necessary letters for victory had been guessed.

C. Integration and End-to-End Testing

Full user journey simulations were conducted to validate the system’s integrated behavior under real-world conditions. These tests focused on the primary user paths and critical edge cases.

There were two prime situations - the success win path and the complete loss path. The win situation utilized the secret word ”debug” and in the pseudo-random play, the system correctly identified the win condition on the last guess, displaying the appropriate congratulatory message. In the complete loss situation, the secret word was ”kernel,” and the game was played with only missed guesses. The testing verified that `chances_left` was decremented correctly for each wrong guess, the hangman ASCII art was updated iteratively, and the game ended with the correct ”Game Over” message and word reveal after `chances_left` was decremented to zero.

Further, edge-case verification again provided confidence in the system. An application test was done using a secret word of 50 characters (the maximum permitted). The system accepted and processed a 50-character word without sufferance of buffer or display issues. To confirm the low-end edge case, the program was tested using a one letter word - ’a’, and correctly registered a win on the first guess. The system was also tested with a user input designed to break the initial word validation for example, empty string, a string with numeric contamination (secr3t), and a string with whitespace (hello world). Each of these failed correctly preventing the user from submitting a word, and to re-enter a valid word.

D. Failure Mode and Security Analysis

The main purpose of the tests was to determine how well the system would withstand typical failure states and often seen vulnerabilities based on user input. In the case of buffer overflow attacks, we tested the program against input longer than the 50-character buffer limit to see how it would handle the input. The `fgets()` function worked as expected by truncating the input and maintaining the validity of the program’s memory because it did not induce a segmentation fault or any kind of corruption. We had also tested for input stream corruption as well. By attempting to guess with multiple characters, we tested that `scanf(" %c")` and the while loop with `getchar()` could isolate the first character of the input, and consume (clear) the remainder while leaving the remaining prompts as valid and unprepared for the invalid input.

This comprehensive testing process confirmed that the application is not only functionally correct across all logical paths but also demonstrates production-grade resilience against invalid inputs, boundary conditions, and common security threats within its defined scope.

VI. Challenges, Solutions, and Future Work

A. Challenges Encountered and Solutions Implemented

The development process presented several engineering challenges that required thoughtful, iterative solutions. The most significant of these was the implementation of secure and reliable user input handling. Initially, the `scanf()` function was considered for its simplicity, but preliminary testing revealed its inherent vulnerability to buffer overflows, a critical security flaw that could crash the program if a user entered an oversized string.

After identifying the risk of using `gets()` , we transitioned to using `fgets()` as a better, safer option. So with `fgets()`’s (like `fgets(secret, MAX, stdin)`) built-in limit on characters to read, there is no risk of an overflow of the buffer. We did find a more subtle area of risk in using `fgets()` since your `fgets` would probably leave the trailing new line character (`n`) and account for that in the string length, or other terms we used for reference, or validation checks. At the end of the day we handled that well with `strcspn()`. With simply `secret[strcspn(secret, "\n")] = '\0';`, we removed the new line virus, and created a data input system that was both safe and tidy.

The second challenge, a little more practically, was to prevent Player 2 from reading the secret word entered by Player 1 on the shared console. Knowing that there are platform specific command such as `system("cls")`, it was determined that those would not be an optimal solution either given the non-portable nature of those commands and the security drawback of the `system()` call. A more efficient and portable solution was used: using a `for` loop to just print 50 newlines so that the secret word was effectively ”scrolled” out of the visible area of the screen. This was convenient since there is guaranteed to be identical, safe behavior on any operating system without having any dependencies on anything external.

In evaluating those solutions, the AI development tools were relied on to help speed up the decisions as a collaborative partner and addressed questions useful for evaluation. For example, they asked questions about the security of input functions; so the comparison of `fgets()` and `strcspn()` essentially confirmed we were using a recommended best practice in the industry and that provided some confidence in the design pattern.

B. Future Work and Potential Enhancements

While the current application is feature-complete within its defined scope, a clear evolution path exists to expand its functionality and architectural sophistication.

Gameplay Enhancements: Future work could focus on increasing replayability and user engagement. A **single-player mode** could be introduced, which would require implementing file I/O operations (using `FILE*` and `fopen()`) to read words randomly from a local dictionary file. This could be coupled with **dynamic difficulty settings**, where the number of chances and the length of the words are adjusted based on user selection (e.g., ”Beginner,” ”Intermediate,” ”Expert”). Furthermore, **thematic word packs** (e.g., ”Programming Terms,” ”Scientific Concepts”) could be added to provide a more tailored gameplay experience.

Architectural Advancements: From a software engineering perspective, the architecture could be significantly improved. First, all individual state variables could be unified into a single `GameState` struct. This would improve maintainability and code clarity by allowing the entire game state to be passed to functions as a single, encapsulated parameter, as shown below.

```
typedef struct
{
  char secret_word[51];
  char used_letters[27];
  int chances_left;
} GameState;
```

Second, the user interface could be upgraded from simple `printf` calls to a more advanced terminal UI using a library like **ncurses**. This would enable features such as color-coded text, dynamic screen refreshing without scrolling, and more sophisticated keyboard input handling.

Performance Optimizations: The current O(N) check for duplicate guesses, while efficient for a small alphabet, could be optimized to a true O(1) operation. This would be achieved by using a simple boolean array as a hash set: `bool guessed[26] = {false};`. A guess could then be checked instantly via its index (`guessed[guess - 'a']`), providing a significant performance gain at scale and demonstrating a more advanced algorithmic approach. This path from a basic CLI application to an enhanced, optimized, and graphically rich program highlights the project’s potential for continued growth.

VII. Conclusion

The project achieved its main goal: creating a functioning, robust and secure two-player Hangman game in a C-based console application. The success of this project is primarily because of a strong defensive input architecture which completely removed buffer overflow risks, and a modular application design to aid readability and ongoing development. The application was thoroughly tested with many forms validity tests, edge cases, and invalid input selections to test stability and functionality, and it was determined to be both stable and secure. The project can be considered a case study in computing professionalism and application of computer science principles, from algorithms to defensive coding, to develop a complete and resilient software product. The final program is demonstrably the application of the accepted software engineering practices within the design, implementation, verification or validation lifecycle.

Acknowledgment

The authors would like to acknowledge the collaborative effort and dedication of all group members that made this project a success. This project was undertaken as a core component of the CSE115L course curriculum at North South University, which provided the foundational knowledge and educational framework necessary for its conception and implementation.

References

[1] C Programming Documentation, GNU Project.
[2] AI Assistance from ChatGPT and Google Gemini