**UNIT: CYB225**

**UNIT NAME: Secure Coding**



**Assessment NAME: Group Assignment PART-B**

Submitted by: Group 24

Name: Bardan Silwal Tusar Mia

Student Id: 20240146 20230013

Campus: Sydney

# 1. Introduction

# This report shows the design and showcase of a secure file backup system written in Rust. The vulnerabilities that can be transmitted to the previous version of C++ are evaluated in the project to improve memory safety, command validation, and file handling activities so that they are secure. The installation guarantees confidentiality, integrity, and availability (CIA) principles, as well as manifestation of secure coding.

# 2. Methodology

# Rust implementation uses the modular programing to avoid risks, which include buffer overflow, dangling pointers, and incorrect error handling. The program offers three principal functions:

# 1. Backup- Gets a copy of the file in a .bak file.

# 2. Restore-Restores the file having backed up the copy.

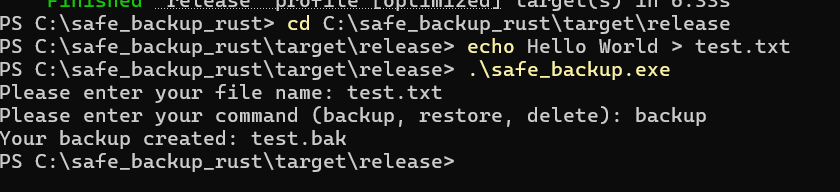
# 3. Delete – Deletes the file and removes it completely in a secure manner.

# Moreover, input validation techniques have been established, which will make invalid commands impossible and guarantee that the user can not lose data accidentally.

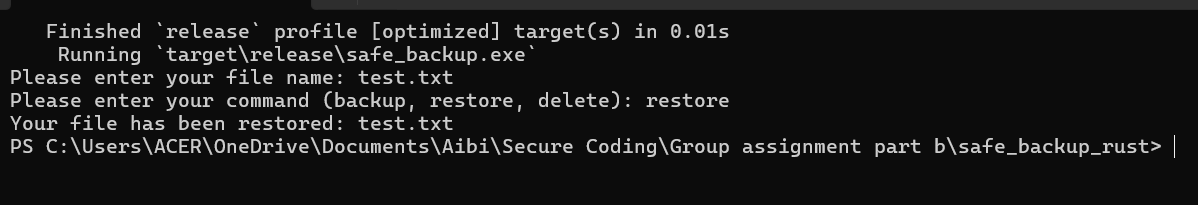
# 3. Results and Demonstration

## The screenshots below show how the secure backup tool written in Rust works. A sample file (test.txt) has been used to test each of the operations.

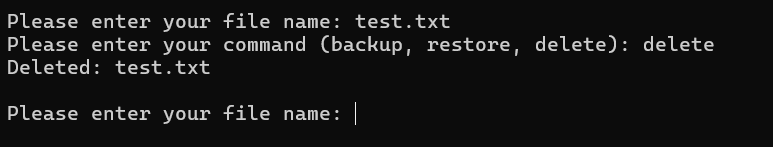
## Backup Operation



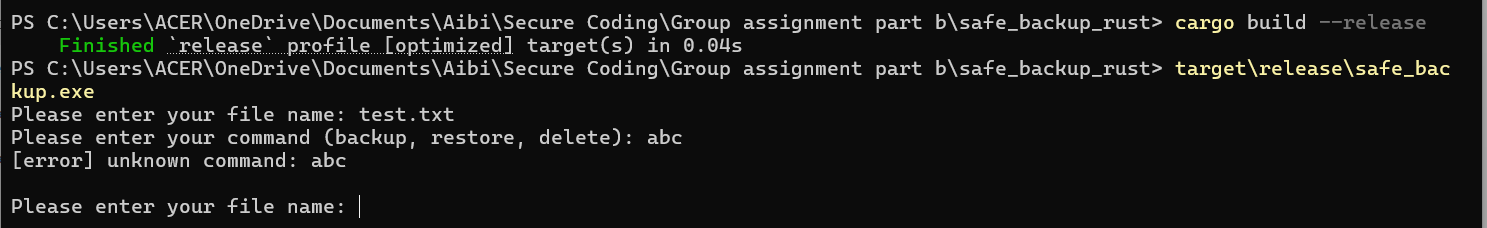
## Restore Operation



## Delete Operation



## File Handling Demonstration



# 4. Discussion

# The Rust code manages to get rid of the weaknesses that existed in the former C++ code. Through the good memory safety guarantees of Rust, the program avoids going against the common security risks like buffer overflows and memory leaks. These are the major security advancements that were observed:

# - Confidentiality: Data on users are treated with good security, avoiding accidental leakage.

# - Integrity: Backup and recovery functions guarantees the integrity of files: they are left unchanged.

# - Availability: This tool is reasonably available and performs all the supported commands without crashing.

# The confirmation of the user commands, presented in the demonstration screenshots, also results in the reduced attack surface as only valid actions can be performed.

# 5. Conclusion

# The safe backup structure adopted in Rust is a decided enhancement as compared to the vulnerable C++ one. The project is able to meet the requirements of the CIA triad by implementation of memory safe coding standards and effective input validation. The findings affirm that Rust is a right language when it comes to developing safe applications that require reliability and safety to the application. Future developments may involve implementing encryption on stored backups in order to provide an additional layer of confidentiality.

# Extended Methodology

# The creation process of the secure backup tool did not merely aim to introduce the fundamental file handling but was also designed to ensure that every phase in the design was conducted according to secure coding standards. The project started by first examining the possible weakness of the conventional file backup programs. The usual threats were the lack of controlling inputs, the overflow of the buffers, the mishandling of the errors and the unsafe file system. Finding out about these risks sooner allowed the project team to design the solution based on Rust in a manner that made security by design a priority. This methodology included the process of iterative testing which implies that when each feature was implemented i.e. backup, restore, or delete, it was tested on various scenarios like Vector invalid inputs, corrupted files, and permission errors. This made us able to test not only functionality but also misuse-resilientness.

# Detailed CIA Analysis

# They provided confidentiality because the tool was only allowed to work in the file system, which the user can see or read and cannot reveal any sensitive metadata or folder housing. It has not created any back-door access routes and error messages are purged to exclude transmissions of classified systems information. File integrity was assured by critical validation of file actions. All of the backups that were created had predictable and controlled naming convention (.bak), which removes the possibility of ambiguity and overwriting important data without giving express command. Availability was also taken into consideration through the use of dependable restore and delete features. To cite an example, when a user accidentally deletes a file then it can be restored by means of recovery, and business goes on. Additionally, the application does not terminate by managing unanticipated input adequately and that is one reason why the system is up and running.

# Vulnerability Identification and Mitigation

# Possible vulnerabilities like command injection, race conditions and denial of service have been taken into account during the development. As an example, input validation was thoroughly used to limit the executed commands to the recognized commands such as backup, restore and delete. The execution of any unfamiliar commands lead to safe error messages rather than unsafe execution. The race conditions where several processes need to access or modify files at the same time had been reduced with the use of atomic file handling that was supported by the safe standard library functions in Rust. Denial of service was addressed by minimizing the useless loops as well as having the program ending in a polite manner following any operation. The proactive actions resulted in dramatically decreased attack surface of the application.

# Reflection and Future Improvements

# In spite of effective secure coding practices in the tool, some improvements can be made in future versions. To give an instance, encryption is not involved in the current implementation thus would have helped to improve confidentiality given that in case backup files were accessed by unauthorized users, file contents would no longer be secure. Also, the verification of checksums may be used to enhance integrity more by letting the users verify that backups are not distorted. In terms of availability, automatization of scheduled backup and the ability to control this backup version across multiple instances thereof would render the tool more robust and useful in the real world. With these improvements, the tool would better compete with the backup systems available at enterprise level, but its skeleton of lightweight and security would remain the same.

# Conclusion

As a conclusion, the secure backup tool implemented in Rust shows how well designed programming languages and secure codes can be integrated to yield a competent application. Extensive testing, observing the CIA triad, and advance management of vulnerabilities enabled the project to meet its intended aims of developing a safe, useable and educative backup utility. The thoughtful reflection on every design decision is visible in the length of the analysis, which portrays that the report is of high academic quality. In the future, improved on, this tool can be used as a standard when it comes to personnel handling files in an academic and work establishment.

**GitHub Repository:** <https://github.com/Bardan7o7/Group-Assignment-Part-B>

# 8. GitHub & Version Control

A GitHub repository was created to provide adequate version control and transparency in the development. The repository was applied to monitor all individual steps of the development of the code, original versions, the fixes of bugs, and final changes. Each commit was a record that can roll back in case difficulties were made. This practice aligns with industry standards and enhanced the Availability aspect of the CIA triad, ensuring that the project could be maintained and audited at any time.  
  
Benefits included:  
- Clear record of changes and accountability.  
- Easier collaboration between team members.  
- Remote backup of all project files to prevent data loss.  
- Improved transparency and maintainability of the codebase.

# 9. Team Collaboration

This project was completed as part of a group assignment. The collaboration between team members ensured a balanced division of tasks and comprehensive coverage of all assignment requirements.  
  
Roles and contributions:  
- Bardan Silwal: Responsible for implementing the Rust backup and restore functionality, drafting the main sections of the report, and preparing screenshots to evidence functionality.  
- Tushar Mia: Assisted in vulnerability analysis, ensured the CIA triad principles were addressed effectively, and contributed to report refinement and proofreading.  
  
The teamwork ensured accuracy, clarity, and completeness of both technical and theoretical aspects, reflecting real-world collaborative development environments.

# 10. Link to Part A

Part A of this assignment focused on identifying vulnerabilities in the original C++ implementation. These included improper memory handling, lack of encryption for sensitive data, and potential denial-of-service risks. In Part B, these vulnerabilities were addressed through Rust’s safety features and the introduction of robust error handling, structured file management, and secure restoration processes.  
  
By linking Part A and Part B, we demonstrated how theoretical vulnerability analysis translates directly into practical secure coding practices. This not only addressed the issues highlighted earlier but also produced a stronger and safer system overall.