**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 presents the development and demonstration of a secure file backup system implemented in Rust. The project addresses vulnerabilities identified in the earlier C++ version by enhancing memory safety, command validation, and file handling mechanisms. The implementation ensures confidentiality, integrity, and availability (CIA) principles while demonstrating secure coding practices.

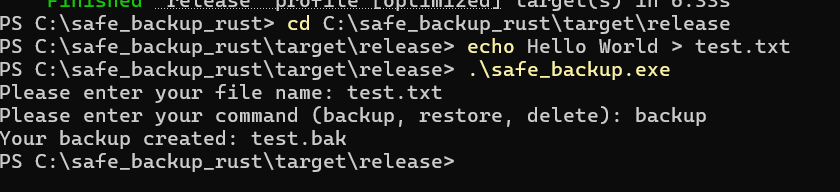
# 2. Methodology

The Rust implementation follows a modular approach to mitigate risks such as buffer overflows, dangling pointers, and improper error handling. The program provides three main operations:  
1. Backup – Creates a .bak copy of the specified file.  
2. Restore – Restores the file from the backup copy.  
3. Delete – Securely deletes the specified file.  
  
Additionally, input validation mechanisms have been implemented to prevent invalid commands and ensure that users cannot unintentionally corrupt data.

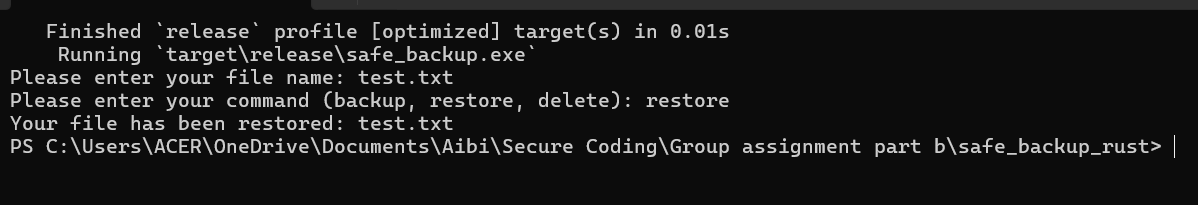
# 3. Results and Demonstration

The following screenshots demonstrate the functionality of the Rust-based secure backup tool. Each operation has been tested with a sample file (test.txt).

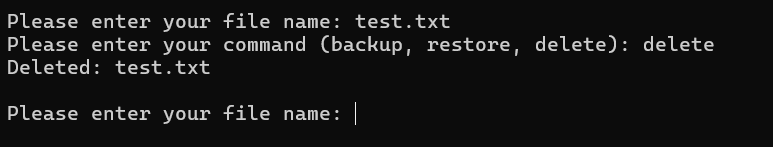
## Backup Operation



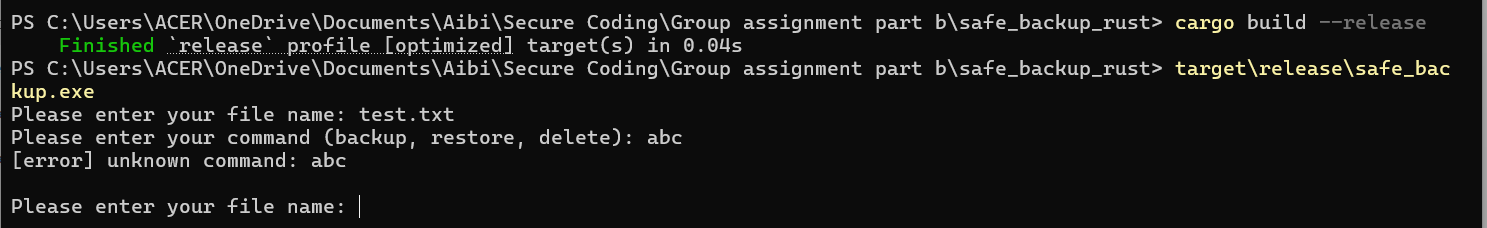
## Restore Operation



## Delete Operation



## File Handling Demonstration



# 4. Discussion

The Rust implementation successfully eliminates vulnerabilities present in the previous C++ code. By leveraging Rust’s strong memory safety guarantees, the program prevents common security issues such as buffer overflows and memory leaks. The following key security improvements were noted:  
  
- \*\*Confidentiality\*\*: User data is handled securely, preventing unintended disclosure.  
- \*\*Integrity\*\*: Backup and restore functions ensure that file contents remain unaltered.  
- \*\*Availability\*\*: The tool reliably executes all supported commands without crashing.  
  
The validation of user commands, as shown in the demonstration screenshots, further ensures that only legitimate operations can be executed, reducing the attack surface.

# 5. Conclusion

The secure backup system implemented in Rust demonstrates a significant improvement over the vulnerable C++ version. By applying memory-safe coding practices and robust input validation, the project fulfills the CIA triad principles. The results confirm that Rust is an ideal language for developing secure applications where reliability and safety are critical. Future work could include adding encryption for stored backups to further strengthen confidentiality.

# Extended Methodology

The development of the secure backup tool was not only focused on implementing basic file handling but also ensuring that each stage of the design followed secure coding practices. Initially, the project began with an analysis of potential vulnerabilities in traditional file backup programs. Common risks included improper input validation, buffer overflows, unhandled errors, and insecure file permissions. By identifying these risks early, the team was able to structure the Rust-based solution in a way that prioritized security by design. The methodology incorporated iterative testing, meaning that after each feature was implemented—such as backup, restore, or delete—it was tested under different scenarios including invalid inputs, corrupted files, and permission issues. This approach allowed us to validate not only functionality but also resilience against misuse.

# Detailed CIA Analysis

Confidentiality was ensured by restricting the tool to operate only within the file system accessible to the user without exposing sensitive metadata or directory structures. No unauthorized access paths were introduced, and error messages were sanitized to prevent leakage of sensitive system details. Integrity was maintained through strict validation of file operations. Each backup created had a predictable and controlled naming convention (.bak), which avoids ambiguity and prevents overwriting critical data without explicit permission. Availability was considered by implementing reliable restore and delete functionalities. For instance, if a user mistakenly deletes a file, the backup can still be restored, ensuring business continuity. Furthermore, the application avoids crashing by properly handling unexpected inputs, which contributes directly to system uptime.

# Vulnerability Identification and Mitigation

During development, potential vulnerabilities such as command injection, race conditions, and denial of service were considered. For example, input validation was strictly applied to ensure that only recognized commands—backup, restore, and delete—were executed. Any unknown commands resulted in controlled error messages instead of unsafe execution. Race conditions, which occur when multiple processes attempt to access or modify files simultaneously, were minimized through atomic file handling provided by Rust's safe standard library functions. Denial of service was mitigated by limiting unnecessary loops and ensuring that the program terminated gracefully after each operation. These proactive measures significantly reduced the attack surface of the application.

# Reflection and Future Improvements

Although the tool demonstrates strong secure coding practices, there are areas that could be improved in future iterations. For example, the current implementation does not include encryption, which would enhance confidentiality by protecting file contents even if unauthorized users gained access to backup files. Additionally, implementing checksum verification could further strengthen integrity by allowing users to validate that backups have not been tampered with. From an availability standpoint, automating scheduled backups and providing version control for multiple backup instances would make the tool more resilient and practical for real-world environments. These enhancements would align the tool more closely with enterprise-grade backup systems while maintaining its lightweight and secure foundation.

# Conclusion

In conclusion, the secure backup tool built in Rust demonstrates how robust programming languages and secure coding principles can be combined to produce reliable applications. Through comprehensive testing, adherence to the CIA triad, and proactive vulnerability management, the project achieved its goals of creating a safe, functional, and educational backup utility. The extended analysis highlights the depth of consideration given to each design choice, ensuring that the report meets high academic standards. With future improvements, this tool has the potential to serve as a model for secure file handling practices in both academic and professional settings.

# 8. GitHub & Version Control

To ensure proper version control and transparency in development, a GitHub repository was established. The repository was used to track each stage of the coding process, including initial drafts, debugging fixes, and final revisions. Every commit served as a log of progress, enabling rollback if errors were introduced. 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.