**Part A**

Read the following papers published on top-tier security conferences:

1. [Learning and Programming Challenges of Rust: A Mixed-Methods Study (songlh.github.io)](https://songlh.github.io/paper/survey.pdf)
2. [2022-78-paper.pdf (ndss-symposium.org)](https://www.ndss-symposium.org/wp-content/uploads/2022-78-paper.pdf)
3. [Rust-lancet: Automated Ownership-Rule-Violation Fixing with Behavior Preservation (songlh.github.io)](https://songlh.github.io/paper/lancet.pdf)

Based on these papers, answer the following questions

1. How does Rust prevent common memory errors, like buffer overflow and use-after-free?
   1. Ownership System:

Rust employs the concept of ownership, where each value in memory has a single "owner" that is responsible for deallocating the memory when the value goes out of scope. This prevents multiple pointers from accessing or modifying the same memory concurrently, thus mitigating the risk of data races and concurrent modification issues.

In our rust code:

*pub struct Maze {*

*pub map: [[i32; COLS]; ROWS],*

*pub start: (usize, usize),*

*pub fin: (usize, usize),*

*}*

In the Maze struct, the ownership of the map field is clearly defined. The map owns the 2D array of integers representing the maze. This ownership ensures that the memory associated with the maze data is deallocated when the Maze struct goes out of scope.

* 1. Borrowing and References:

Borrowing and References: Instead of relying solely on ownership, Rust introduces the concept of borrowing, allowing multiple immutable references or one mutable reference to a value. These references are checked at compile time to ensure that they do not outlive the data they refer to or conflict with other references. This prevents both dangling pointers and data races.

From out rust code:

*fn get\_walkable\_neighbors(&self, point: VisitedCell) -> Vec<VisitedCell> {*

*// Function taking immutable reference to self (borrowing)*

*// and returning a vector of VisitedCell references*

*}*

The **get\_walkable\_neighbors** function takes an immutable reference to self, allowing it to borrow the Maze instance. This ensures that **get\_walkable\_neighbors** cannot modify the Maze instance, preventing concurrent modifications. Additionally, the function returns a vector of VisitedCell references, which borrow data from the Maze instance rather than taking ownership.

* 1. Lifetime Annotations:

Rust's borrow checker analyzes the lifetimes of references to determine their validity. Developers can use lifetime annotations to explicitly specify the relationships between references, helping the compiler verify that references do not outlive the data they point to or reference data after it has been deallocated.

* 1. No Null Pointers:

Rust does not have null pointers or null references by default. Instead, it uses an optional type called Option<T> to represent optional values. This eliminates the possibility of null pointer dereferencing errors, as the compiler ensures that values are properly initialized before use.

In our code, we can see rust avoids null pointers by using the Option type. In the load\_grid function, file is of type Option<File>. Instead of checking for null pointers, Rust uses the Option enum to represent the presence or absence of a value. Here, file.is\_none() checks if the file variable contains a valid file handle.

* 1. Bounds Checking:

Rust performs bounds checking on array accesses at runtime to prevent buffer overflow errors. This ensures that accesses to array elements are within the bounds of the array, preventing memory corruption due to invalid memory accesses.

Rust performs bounds checking to prevent buffer overflow errors. In the get\_walkable\_neighbors function, nx is checked against the bounds of the maze rows (ROWS) to ensure that array accesses stay within valid bounds. This prevents accessing memory outside the allocated array.

For example:

*if nx < 0 || nx >= ROWS.try\_into().unwrap() {*

*continue;*

*}*

* 1. Unsafe Rust:

While Rust emphasizes memory safety, it also provides an unsafe keyword for opting into low-level operations that bypass the safety checks of the compiler. However, the use of unsafe blocks is confined to specific portions of code where safety invariants can be manually upheld by the developer, and such code is subject to rigorous review and testing.

Unsafe Rust allows bypassing some of the safety checks of the compiler. In this example, ***unsafe*** is used to push a value into the ***been\_to*** vector without performing bounds checking or lifetime validation. However, the use of ***unsafe*** should be minimized and contained within well-audited blocks.

Example:

*unsafe {*

*been\_to.push(child.coord);*

*}*

1. What are the remaining security issues in rust programs?
   1. Concurrency and Data Races

*let mut been\_to: Vec<(usize, usize)> = Vec::new();*

Although Rust's ownership system prevents data races at compile time, concurrent access to mutable data can still lead to logic errors. In this example, been\_to is a mutable vector that could be accessed concurrently by multiple threads. To mitigate this, we should use Rust's concurrency primitives like Mutex or Arc to enforce thread safety.

* 1. Unsafe Code:

Rust allows bypassing safety checks using the unsafe keyword. While necessary for interfacing with low-level system APIs or optimizing performance-critical code, misuse of unsafe can introduce vulnerabilities like buffer overflows or null pointer dereferences. It's crucial to use unsafe judiciously and carefully audit such code.

*unsafe {*

*been\_to.push(child.coord);*

*}*

* 1. Fuzzing and Input Validation:

For example: *let file = fs::read\_to\_string(file\_name).unwrap();*

Reading input from external sources like files or network streams without proper validation can lead to security vulnerabilities like buffer overflows or injection attacks. In this example, read\_to\_string assumes that the file exists and can be read, potentially leading to unexpected behavior or vulnerabilities. Input should be thoroughly validated and sanitized to prevent exploits.

* 1. Memory Safety Bugs in External Dependencies:

For example: *let lines = file.lines();*

Rust programs often rely on external dependencies, which may contain memory safety bugs. While Rust's safety guarantees apply to safe Rust code, unsafe code within dependencies could introduce vulnerabilities. It's essential to vet dependencies for security issues and keep them updated to minimize risks.

* 1. Memory Leaks and Resource Exhaustion:

For example: *return Some(child);*

Improper resource management can lead to memory leaks or resource exhaustion. While Rust's ownership system mitigates many memory-related issues, incorrect handling of resources like file handles or network connections could still occur. Proper resource management, such as using RAII patterns or Rust's Drop trait, is essential to prevent these vulnerabilities. [4]

1. What are the main challenges of programming in rust?
   1. Ownership and Borrowing:

In Rust: *let mut been\_to: Vec<(usize, usize)> = Vec::new();*

Compared to C: *std::vector<Coord \*> been\_to;*

In Rust, managing ownership and borrowing can be challenging, especially when dealing with complex data structures like vectors or structs. Rust's ownership system ensures memory safety by enforcing strict rules on ownership transfer and borrowing, which may require careful consideration and restructuring of code compared to the more relaxed memory management in C.

* 1. Concurrency and Mutable State

In rust: *let mut queue: VecDeque<VisitedCell> = VecDeque::new();*

In C: *std::deque<Coord \*> q;*

Rust encourages safe concurrency through its ownership and borrowing system, which helps prevent data races and mutable state issues at compile time. However, managing concurrent data structures and ensuring thread safety can be challenging compared to C, where developers have more flexibility but must manually synchronize access to shared data.

* 1. Safety vs. Performance Trade-offs:

In Rust: *unsafe {*

*been\_to.push(child.coord);*

*}*

In C: *been\_to.push\_back(child);*

Rust's safety guarantees come with a performance cost, particularly when using unsafe code blocks to bypass safety checks. While Rust aims to provide safe abstractions without sacrificing performance, developers may need to carefully balance safety and performance considerations, especially in performance-critical sections of code, compared to C, where performance optimizations are more manual but potentially more straightforward.

* 1. Learning Curve:

Rust introduces new concepts like ownership, borrowing, lifetimes, and pattern matching, which may require developers to invest time in understanding and mastering these concepts compared to the more straightforward syntax and semantics of C. While Rust's safety features provide benefits in terms of reliability and maintainability, they also require a learning curve for developers transitioning from languages like C.

For Example:

In Rust: *let mut fin: (usize, usize) = (0, 0);*

In C: *int fin[2] = {0, 0}; [5]*

1. How can we make rust easy to use?
   1. Improved Documentation and Learning Resources: Providing clear and comprehensive documentation is essential for reducing the learning curve associated with Rust. Alongside documentation, tutorials, guides, and examples play a crucial role in helping developers understand the language's concepts and best practices. Detailed explanations, code samples, and real-world use cases contribute significantly to enhancing understanding and facilitating usage, especially for complex concepts that may be challenging to grasp initially.
   2. Ergonomic Syntax and Abstractions: Rust's syntax prioritizes clarity and expressiveness, aiming to enhance readability and ease of use compared to more verbose languages. The language offers ergonomic abstractions and data structures that reduce boilerplate code and simplify common tasks. These abstractions not only make code more concise but also improve developer productivity by streamlining the development process.
   3. Tooling and IDE Support: Rust's build system, Cargo, along with tools like Rust Analyzer and Rustfmt, provides a seamless development experience for programmers. Robust tooling and IDE support, including features such as code completion, linting, and refactoring, significantly enhance developer productivity and ease of use. These tools not only aid in writing high-quality code but also assist developers in navigating and understanding Rust's features and libraries.
   4. Community Engagement and Support: Rust boasts an active and welcoming community that fosters support and engagement among developers of all skill levels. Forums, chat rooms, and community events provide valuable platforms for developers to seek help, share knowledge, and collaborate on projects. The supportive nature of the Rust community encourages learning and exploration, making Rust more accessible and inclusive to a wide range of developers.
   5. Incremental Learning and Adoption: Encouraging incremental learning and adoption of Rust is essential for developers to grasp the language's concepts gradually. Providing small, manageable projects and tutorials allows developers to build confidence and familiarity with Rust over time. Starting with simple examples and gradually introducing more advanced features helps developers gain practical experience and solidify their understanding of Rust's capabilities. [6]
2. What is the main idea of rust-lancet? Do you think it is useful for rust developers and why?

Rust-lancet is a tool that focusses on ownership-rule-violations, as the way it works it takes a program with safety-rule violations as an input and proceeds to either produce a modified program that does not contain any violations and preservers the semantics of the original program, or it will report that it cannot successfully patch the program following several attempts [3].

In essence, the idea of Rust-lancet is to automate the process of fixing the safety violations that exist within a programmer's code, as the safety rules are as strict as they are hefty. This tool can indeed be useful, so long as it is not solely relied upon, as to put blind faith in this tool and expect a given program to truly be both violation free and fully functional as intended by the programmer 100% of the time would be absurd.

Another point to consider is that the tool is used to assist the optimization not only the work, but also the workflow for the programmer, so should the tool continuously fail after constant tweaking, it may have been more efficient for the programmer to have manually fixed the safety-rule violations than not in the essence of time and peace of mind.

In summation, as with most tools that are meant to automate tasks, they are best used and tweaked within the means of their use cases, and even regardless of the use case, they should always be monitored and checked for correctness to ensure optimal results.

**References**

[1] S. Zhu, Z. Zhang, B. Qin, A. Xiong, and L. Song, “Learning and programming challenges of rust,” *Proceedings of the 44th International Conference on Software Engineering*, May 2022, doi: <https://doi.org/10.1145/3510003.3510164>.

‌[2] S. Mergendahl, N. Burow, and Hamed Okhravi, “Cross-Language Attacks,” Jan. 2022, doi: <https://doi.org/10.14722/ndss.2022.24078>.

‌[3] W. Yang, L. Song, and Y. Xue, “Rust-lancet: Automated Ownership-Rule-Violation Fixing with Behavior Preservation,” *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Apr. 2024, doi: <https://doi.org/10.1145/3597503.3639103>.

[4] Bugden, William, and Ayman Alahmar. "Rust: The programming language for safety and performance." arXiv preprint arXiv:2206.05503 (2022).

[5] Zhang, Yuchen, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. "Towards understanding the runtime performance of rust." In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1-6. 2022.

[6] Fulton, Kelsey R., Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. "Benefits and drawbacks of adopting a secure programming language: Rust as a case study." In Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021), pp. 597-616. 2021.