applied sciences

MDPI

Article

Thetis: A Booster for Building Safer Systems Using the Rust Programming Language

Renshuang Jiang [1,†], Pan Dong [1,†], Yan Ding [1], Ran Wei [2] and Zhe Jiang [3,*]

[1] College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China; renshuang717@163.com (R.J.); pandong@nudt.edu.cn (P.D.); yanding@nudt.edu.cn (Y.D.)
[2] Department of Engineering, University of Cambridge, Cambridge CB2 1PZ, UK; rw741@cam.ac.uk
[3] Department of Computer Science and Technology, University of Cambridge, Cambridge CB3 0FD, UK
* Correspondence: zj266@cam.ac.uk
† These authors contributed equally to this work.

Abstract: Rust is a new system-level programming language that prioritizes performance, safety, and productivity. However, as evidenced in many previous works, unsafe code fragments broadly exist in Rust projects. The use of these unsafe fragments can fundamentally violate the safety of systems developed using the programming language. In response to this problem, we propose a novel methodology (Thetis) to enhance the safety capability of Rust. The core idea of Thetis is to reduce unsafe code, encapsulate unsafe code using safety rules, and make it easier to verify unsafe code through formal means. The proposed methodology involves three main components. In the context of Rust itself, Thetis combines replacement and encapsulation for *Interior Unsafe* segments, minimizing unsafe fragments and reducing unsafe operations and their range. For systems developed using Rust, new ACSL formal statutes are applied to reduce the unsafe potential of the encapsulated *Interior Unsafe* segments, enhancing the safety of the system. Regarding the development life cycle in Rust, Thetis introduces automatic defect detection and optimization based on feature extraction, improving engineering efficiency. We demonstrate the effectiveness of Thetis by using it to fix defects in BlogOS and ArceOS. The experimental results reveal that Thetis reduces the number of unsafe operations in these OSs by 40% and 45%, respectively. The use of Miri to detect and eliminate defects in ArceOS reduces the likelihood of undefined behavior by about 50%, which effectively demonstrates that the proposed method can improve the safety of the Rust system. In addition, performance test results from LMbench show that the performance loss caused by Thetis is only 1.076%, thereby maintaining the high-performance characteristics of the Rust system.

Keywords: unsafe fragments; interior unsafe; static analysis; system safety

Citation: Jiang, S.; Dong, P.; Ding, Y.; Wei, R.; Jiang, Z. Thetis: A Booster for Building Safer Systems Using the Rust Programming Language. *Appl. Sci.* **2023**, *13*, 12738. https://doi.org/10.3390/app132312738

Academic Editor: Paolino Di Felice

Received: 15 October 2023
Revised: 18 November 2023
Accepted: 20 November 2023
Published: 28 November 2023

1. Introduction

Traditional system-level programming languages often suffer from memory and concurrency safety issues [1–4], whereas memory-safe programming languages such as GO usually suffer from significant runtime overhead [5]. Rust, the most popular system-level programming language [6,7], balances the two scenarios through its unique safety mechanisms such as ownership and lifetime [8,9]. These mechanisms prevent many vulnerabilities caused by memory accesses (e.g., use-after-free and buffer overflow) and thread concurrency (e.g., data competition). Essentially, Rust aims to ensure safety akin to high-level languages and performance efficiency comparable to low-level languages, making it an increasingly popular choice for building fundamental software (e.g., operating systems (OSs) and browsers [10,11]). For example, the rewriting of the Firefox kernel in Rust achieved better parallelism, safety, modularity, and performance [12].

However, despite the safety mechanisms, existing projects written in Rust may still have many serious software vulnerabilities. For example, hundreds of bugs have been

identified in the RustSec Advisory Database [13], including rulex bugs, static-type-map unmaintained bugs, etc. The main reason for this is that Rust projects for system-level operations usually contain some unsafe code fragments, which are necessary for low-level operations but lack the safety guarantees provided by the compiler. More specifically, Rust comprises two distinct programming languages: Safe Rust (denoted by safe code) and Unsafe Rust (denoted by unsafe fragments, as the number of code lines is small). Safe Rust is recommended more by the developers of Rust, as it enables the writing of high-performance applications and libraries using safety rules, so this type of Rust code constitutes the main body of the project. On the other hand, unsafe fragments distinguish themselves using the keyword "unsafe" to bypass strict compiler checks to support more low-level controls. Unsafe fragments are mainly used to implement five types of operations [14]: (1) dereferencing a raw pointer; (2) calling an unsafe function or method; (3) accessing or modifying a mutable static variable; (4) implementing an unsafe *trait*; and (5) accessing fields of a *union*. Previous research has verified that a program fully written in safe code can completely avoid memory errors [13,15,16]; however, the unavoidable unsafe fragments may introduce vulnerabilities and make the system less safe. Although Rust does not completely address the safety issues of system-level programming, it still offers many advantages over traditional languages such as C.

How can the safety of a system be improved when unsafe fragments are unavoidable? Formal verification is a commonly used approach, which verifies a system by exploring its state space using mathematical methods [17]. However, for operating systems, the amount of code to be verified is huge, the boundaries are extensive, and the code dependencies between kernels are high. The overhead of verification is not affordable and heavily depends on the experiences of the engineers. Therefore, we focus on minimizing unsafe fragments to reduce the unsafe state space, which requires formal verification to ensure safety. To this end, in this paper, we identify three key challenges:

- Only formal verification approaches can completely eliminate the vulnerabilities introduced by unsafe fragments. However, the complexity and dependencies within operating systems lead to an explosion of formal verification state spaces [18–20]. This results in formal methods validating only a subset of the system, which is the main reason for the low feasibility of formalization. To address this problem, new methods are needed to reduce the set of states in the verification space, thereby reducing the amount of code and interactions between code.
- Unsafe fragments in Rust can fundamentally compromise the safety of developed software, especially OSs. This is the major reason for memory and thread issues and is unavoidable in system implementations. Therefore, to address this issue, new methods are required to minimize the use of unsafe fragments and ensure the safety of developed software. These methods also need to ensure safety for unsafe fragments that cannot be eliminated.
- Although unsafe fragments are a major contributing factor to system vulnerabilities, interactions between safe and unsafe code also lead to issues such as memory life-cycle (MLC) bugs [21]. These interactions can cause misunderstandings and the improper use of lifetimes. Due to unavoidable unsafe fragments, a system cannot avoid these interactions. To address this issue, new methods are required to reduce the arbitrary interactions between safe and unsafe code.

To address these challenges, we propose the Thetis methodology. The core of this approach is to minimize unsafe fragments and interactions. We have gleaned two key factors derived from facts. The first is that there are only five types of unsafe code [14] and the second is that using *Interior Unsafe* segments can effectively improve safety and compress the state space to be formally verified. For a limited number of five unsafe operations, Thetis first replaces them with safe functions. Then, it encapsulates other operations, which cannot be replaced, as *Interior Unsafe* functions. Furthermore, combined with static analysis, new ACSL (the ANSI/ISO C Specification Language) formal specification constructs are designed for the encapsulated functions. Based on these approaches, Thetis simplifies the

implementation and reduces the unsafe state space. Compared to similar technologies, the Thetis method also forms a log of safety checks through automated feature detection. It offers a significant advantage in terms of improving the efficiency and scalability of system development. The results of the automated detection can also be used as input for techniques such as formal verification to improve efficiency and effectiveness. By using the defect detection method proposed in this article for defect detection and classification, the accuracy can reach 84.6%. To demonstrate the effectiveness of Thetis, we use it to fix defects in BlogOS and ArceOS. The experiment shows that the number of unsafe operations in these OSs was reduced by about 40% and 45%, respectively. Thetis greatly reduced the scope of unsafe code. In order to better observe the improvement in the security performance of the Rust system, we used the Miri detection method for ArceOS before and after defect elimination, reducing the likelihood of undefined behaviors by about 50%. In addition, in this paper, Thetis only achieved an overall performance loss of 1.076%.

The rest of this paper is organized as follows. Section 2 outlines the need for unsafe superpowers in Rust coding and reviews the related works on safety improvements. Section 3 describes the design idea of the Thetis approach based on minimizing unsafe operations. Section 4 details the design and construction of Thetis. Section 5 presents the test results of Thetis for both security and performance aspects. Section 6 concludes with a summary and an outlook for future work.

2. Background

2.1. Rust Language

High performance, safety, and productivity are the main features of Rust [16]:

- High performance: Rust's zero-cost abstraction, high memory efficiency, and no runtime garbage collection make it a high-performance programming language, suitable for the development of OSs and applications.
- Safety: The ownership model ensures Rust's safety by borrowing rules and a rich-type system in safe code. Many vulnerabilities that are difficult to find in C can be found in Rust's compile time, which provides better memory and concurrency safety, providing more manageable and reliable error handling.
- Productivity: Productivity is achieved through well-documented specifications, a user-friendly compiler, clear error messages, and other helpful tools and resources. These features help the developers understand and use the language, leading to increased productivity in the development process.

Rust systems can be considered as consisting of safe code and unsafe fragments. Safe code is the recommended way to write Rust code, ensuring that the developer does not have to worry about undefined behaviors [22–24]. Rust's safety mechanism is built around the concepts of ownership and lifetime to prevent memory and concurrency errors. Rust ownership [9] states that "each value in Rust has an owner; there can only be one owner at a time, and when the owner goes out of the scope, the value will be dropped". Overall, a safe mechanism can efficiently analyze the use of memory resources during the compilation phase, enabling fine-grained management of memory.

Unsafe fragments are exactly like safe code with all the rules and semantics, but they do not have the same memory safety guarantees. Unsafe fragments exist because, by nature, static analysis is conservative. When the compiler does not have enough information to guarantee the code's safety, the code may be rejected. Another reason Rust has an unsafe alter ego is that the underlying computer hardware is inherently unsafe [14]. Programmers usually use the keyword "unsafe" to bypass compiler checks and perform some operations that may be vulnerable, which provides more flexibility for Rust. However, unsafe fragments are similar to C, suffering from many memory and thread issues. Previous research shows that when checked by the compiler, programs fully written in safe code can completely avoid memory errors [15]; thus, unsafe fragments are the major reason for the reduction in the safety of OSs. Therefore, the issue of Rust safety is focused on the

improvement of unsafe fragments. The following subsections analyze the specific reasons and the necessity of using unsafe fragments.

2.2. Unsafe Rust

The official Rust documentation [14] specifies five actions that are allowed in unsafe fragments but not in safe code, which are called unsafe superpowers:

1. Dereference a raw pointer
   Unlike smart pointers, raw pointers are not restricted by safety mechanisms. They cannot guarantee that the memory addresses they point to are always valid, and they also lack the concept of lifetime. Additionally, raw pointers require manual memory management, similar to C, rather than automatic memory management, which is provided by smart pointers. Therefore, the compiler is unable to check for proper borrowing or safety guarantees when using raw pointers. Creating a raw pointer itself is not harmful, but attempting to access the memory address that it points to may return an invalid value, which is an unsafe operation.
2. Call an unsafe function or method
   Unsafe fragments are often introduced by calling unsafe functions or methods, leading the compiler to no longer be able to verify the safety issues, e.g., illegal memory accesses. Therefore, these unsafe functions and methods must be placed in an unsafe code block to prevent possible issues. However, the unsafe function code block introduces many unsafe operations. It expands the scope of unsafe fragments, and other bugs may be inadvertently introduced, which makes it more difficult to ensure safety. We show an example in Listing 1. The active_level_4_table() function needs to be implemented for memory management.

   Listing 1. Active 4-level page table references.

```
unsafe fn active_level_4_table(physcial_memory_offset:
    VirtAddr)-> &'static mut PageTable {
     let (level_4_table_frame, _) = Cr3::read();
     let phys = level_4_table_frame. start_address();
     let virt = physcial_memory_offset + phys.as_u64();
     //call as_mut_ptr() to convert it to a raw pointer
     let page_table_ptr: *mut PageTable = virt.
        as_mut_ptr();
     //dereference a raw pointer
     &mut *page_table_ptr;
}
```

   This function creates a raw pointer using the as_mut_ptr() method, which is a safe operation. Dereferencing the raw pointer in this function requires separate safety guarantees. Therefore, it should be defined as an unsafe function.
3. Implement an unsafe trait
   A Rust *trait* is similar to an interface in traditional languages like *Java*. A *trait* is considered unsafe when it contains at least one method with an invariant that the compiler cannot verify. Its safety needs to be individually guaranteed by the programmer. Simply put, the vulnerability of an unsafe *trait* arises from unsafe functions or methods.
   A GlobalAlloc *trait* can be used to implement a heap allocator, as shown in Listing 2. The compiler cannot check this automatically, instead requiring the programmer to ensure the truth of the allocator type.

Listing 2. Implementation of GlobalAlloc traits.

```
pub unsafe trait GlobalAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;
    unsafe fn dealloc(&self, ptr: *mut u8, layout:
       Layout);
    // ... other functions
}

unsafe impl GlobalAlloc for SimpleAllocator{ //... }
```

For example, the alloc() method in this *trait*, must request an unallocated memory block; otherwise, it may result in undefined behaviors. Therefore, the alloc() method is unsafe, which also makes the GlobalAlloc *trait* unsafe.

4. Access or modify a mutable static variable
   Accessing or modifying immutable static variables does not result in any safety issues. However, for mutable static variables, if the same variable is accessed by multiple threads, it will cause data competition and data safety cannot be guaranteed. As shown in Listing 3, accessing the mutable static variable STACK requires putting the operation into the unsafe block.

Listing 3. Access or modify a mutable static variable.

```
static mut STACK:[u8; STACK_SIZE] = [0;STATCK_SIZE];
let stack_start = VirtAddr::from_ptr(unsafe {&STACK});
```

5. Access fields of a union
   During memory initialization, the *union* only specifies the value of one field within it. Therefore, accessing the fields of the *union* may result in accessing undefined fields. This leads to undefined behaviors, which the compiler cannot check. Therefore, it is unsafe to read and write *union* fields.

Rust's safety rules are strict, and the compiler's static checking of these rules is conservative. This leads to unsafe code fragments in many projects, which conflicts with the safety of the system. Balancing these two factors is a challenging problem.

2.3. Rust Language Safety Improvement Study

To reduce the vulnerabilities caused by unsafe fragments, researchers have conducted various studies, including code suggestions/specifications [15,25,26], formal verification [27–29], fuzzy testing [30–32], symbolic execution [33–35], and static analysis [36–38].

Zhu et al. [15] proposed six suggestions for learning and programming Rust, allowing programmers to better understand Rust's safety mechanisms and develop safer systems. Qin et al. [25,26] performed the first empirical study of Rust and proposed an important concept known as *Interior Unsafe*. *Interior Unsafe* is defined as a pattern with a function (or an interface implementation) containing a piece of unsafe code, while the function (or interface itself) can be called safe. Regarding *Interior Unsafe*, two helpful insights were mentioned in the paper: (1) *Interior Unsafe* is a good way to encapsulate unsafe code, and developers should first try to properly encapsulate unsafe code in *Interior Unsafe* functions before exposing them as unsafe, and (2) *Interior Unsafe* functions often rely on the preparation of correct inputs and/or execution environments for their internal unsafe code to be safe. Using std::sync::Mutex as an example [39] (see Listing 4), the mutex provides an encapsulated lock() API, adopting the *Interior Unsafe* concept. Although these studies provide suggestions for building safer Rust coding specifications, they are all empirical studies, and the approaches therein depend on the developer's own familiarity with Rust and development experience. They do not effectively relieve the burden on developers.

Listing 4. *Interior Unsafe* functions in the standard library.

```
impl<T: ?Sized> Mutex<T> {
    pub fn lock(&self) -> LockResult<MutexGuard<'_, T>>{
        //Interior Unsafe function
         unsafe {
            self. inner. raw_lock();
            MutexGuard::new(self)
        }
    }
}
```

Formal verification is a method of proving the correctness of operating systems using theorems. For example, RustBelt [27] is the first formal safety proof for Rust, but it can only verify a subset rather than all of Rust's features. Li et al. [37] proposed formalization as a way of detecting defects, but since the amount of code is so large, it cannot detect all defects. QIAN et al. [40] proposed that due to kernel function dependencies and the global nature of verification, formal verification is very difficult. They also pointed out that verifying one line of C code requires, on average, about twenty-five lines of proof code. Therefore, as code size and dependencies increase, the complexity of verification increases, requiring a lot of manual work to perform the transformation, which reduces the efficiency.

Miri [41] detects common undefined behavior in Rust code, such as out-of-bounds memory accesses, use of uninitialized data, and data races, by generating a Rust intermediate representation called MIR. However, using this tool for detection results in a significant number of false negatives and false positives.

The fuzzy testing tool libFuzzer [30] requires analysis of the situation to design fuzzing targets, making it unsuitable for direct use in library APIs. LibFuzzer was further improved by RULF [31] but still infers specific types of dependencies. Therefore, this method is unsuitable for trait operations. Solutions like symbolic execution methods [33,34] also suffer from the same issues. Rudra [36] detects memory issues in unsafe fragments through static analysis. However, it can only analyze certain problems.

According to Rice's theorem [42], any nontrivial property of a language recognized by a Turing machine is undecidable. This means that a program can be considered a function that maps inputs to outputs, and there is no generic algorithm that can detect the non-trivial properties (i.e., properties that are not true or false for all programs) of this function. Therefore, there is no perfect static analysis that addresses all safety issues. In the static analysis process, there is inevitably sound and complete behavior [43], as shown in Figure 1. Thus, the existing Rust static analysis tools [36–38] all suffer from false negatives and false positives.

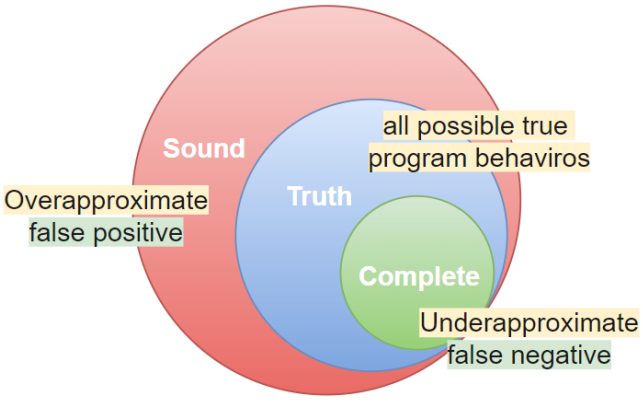

Figure 1. Static analysis of false positives and false negatives.

The existing studies for improving the Rust language do not solve the problem efficiently, safely, and easily. In summary, since the compiler cannot ensure the safety of unsafe fragments, formal methods are the only tools that can prove that they are truly safe. However, the complexity and dependencies of the code make the current proofs difficult and only a subset of them can be proved. It is a challenge not only to minimize the use of unavoidable unsafe fragments and reduce the interactions between safe and unsafe code, which reduces the complexity of Rust unsafety and formal verification, but also to automate the process to achieve greater efficiency.

3. Thetis Design Ideas

As discussed in Section 1, the amount of unsafe code and dependencies are the main causes of unsafe systems and formal verification difficulties. Enhancing the safety capability of Rust and systems developed using Rust necessitates the implementation of a new methodology to (i) limit the use of unsafe fragments, and (ii) minimize the occurrence of unplanned interactions between safe and unsafe code.

Regarding (i), we have observed that the specification-like solutions [15,25,26] introduce a lot of subjective randomness, the test-like solutions [33,37,38] all lack accuracy due to the existence of false positives and false negatives, and the formal-like solutions [27,31,34] mostly prove a subset of the language due to state explosions and heavily depend on a lot of experts' efforts. Therefore, the key idea for addressing this challenge is to balance the trade-off between test efficiency and proof complexity by minimizing the use of unsafe code as much as possible. The limited types of unsafe code in Rust provide a means to achieve this.

Regarding (ii), the Rust programming specification suggests encapsulating unsafe operations with a safe interface [15,25,26]. *Interior Unsafe* functions are an effective way to realize this. Using *Interior Unsafe* functions to encapsulate unsafe fragments may provide a safe calling interface for developers. It significantly reduces the scope of unsafe fragments and the interactions between unsafe and safe code. These implementations are generally subjected to rigorous manual checks, so the safe code interfaces built on top of these implementations can be assumed to be safe. We note the second insight about *Interior Unsafe* functions in [25] and develop a novel idea to use ACSL to enhance their safety. This idea partially addresses the problem of the preparation of correct inputs and/or execution environments, and it can also reduce the state space of the formal verification.

With this in mind, we propose a new methodology, known as Thetis. The overall design of Thetis is based on the idea of minimizing the use of unsafe fragments and follows four principles:

- **Principle 1: Functionally equivalent replacement.** For finite-type unsafe operations, Thetis first replaces them with functionally equivalent safe Rust functions where possible. This principle directly eliminates the vulnerabilities of unsafe fragments.
- **Principle 2: *Interior Unsafe* encapsulation.** For unavoidable unsafe operations, Thetis encapsulates those that cannot be replaced within *Interior Unsafe* functions. This principle reduces the scope of unsafe effects, thus indirectly reducing the vulnerabilities of unsafe fragments.
- **Principle 3: Adding the ACSL formal specification.** New ACSL formal specifications are presented to constrain the potential vulnerabilities of *Interior Unsafe* encapsulated functions. This principle provides lifetime support and partial verification to the variables or operations in the *Interior Unsafe* section.
- **Principle 4: Automatic defect detection and revision.** Thetis introduces an automatic detection technique for extracting defects based on unsafe features, and the generated defect report provides guidance for minimizing unsafe fragments. Thetis can automatically provide the suggested safe code to replace the unsafe code, with the same functionality.

The basic Thetis framework is shown in Figure 2. Specifically, Thetis manages five types of unsafe operations in a flexible, safe, and efficient manner, as discussed in Section 2.

Thetis is designed based on a finite type of unsafe operations, reducing the complexity of optimization and the possibility of false positives and false negatives and ensuring that the system is as safe and reliable as possible. Firstly, it tries to directly replace the mutable static variables and *union* types with safe implementations to eliminate unsafe fragments.

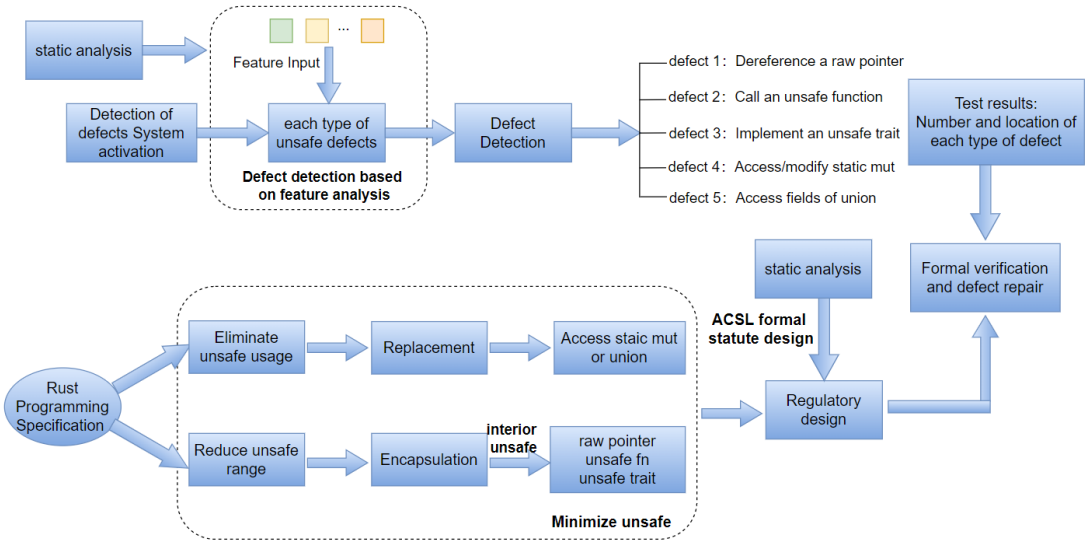

Figure 2. Basic framework of Thetis.

Before diving into the specific design details, we reiterate the necessity and reasons for the replacement. Like the designers of Rust, we also consider that union-type design is necessary for Rust's language design. It provides interoperability with low-level system programming and reduces memory storage requirements. In certain situations, it bypasses compiler checks, offering greater flexibility and efficiency. Additionally, in cases where safety is the primary concern, we may be willing to sacrifice some performance and programming flexibility for the sake of safety.

For operations that are not replaceable, such as dereferenced raw pointers, unsafe functions, and *traits*, the Rust programming specification suggests encapsulating unsafe operations with a safe interface [15,25,26]. Therefore, Thetis encapsulates these operations based on the concept of *Interior Unsafe*. Developers only need to call these safe interfaces when designing the system. This effectively reduces the scope of unsafe fragments and the interactions between unsafe and safe code, improving code maintainability. For systems developed using Rust, Thetis combines the ACSL static analysis methods for *Interior Unsafe* encapsulated functions to formalize the specification. It provides lifetime support and partial verification for unsafe fragments. For example, asserts are used to verify that unsafe fragments satisfy certain properties, and function contracts are used to verify that implementations and invocations of functional modules are legal. This method provides a framework to help reduce common unsafe states and potential vulnerabilities in the system. The combination of the two approaches minimizes unsafe fragments and reduces the amount of code checking required. Developers only need to call these designed safe interfaces in use, which reduces the burden on developers. With the scope of unsafety reduced, the state space of the formal proof becomes easier to analyze. This method balances testing efficiency and proof complexity, which allows systems to be safely verified using formal methods. Finally, considering that manual defect analysis is time-consuming and difficult, Thetis uses lexical analysis for finite types of unsafe operations and offers automatic defect checks and optimization based on unsafe features. The detection results form a log that can provide guidance for formal verification, thus improving efficiency and effectiveness.

4. Design and Construction of Thetis

In the specific implementation of Thetis, our goal is to address all five types of unsafe operations to reduce system-wide vulnerabilities with the guidance of the aforementioned principles.

4.1. Functionally Equivalent Replacement

4.1.1. Replacement of Mutable Static Variables

Static variables are placed in a data segment at initialization, and the data in that space occupy that storage for the entire runtime of the program. When multiple threads access the same mutable static variable, concurrency issues may occur, including Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW) issues. The most common problem is data competition, which may cause deadlocks and result in invalid data.

Thetis detects mutable static variable usage based on features and replaces them with mutexes. It can directly transform the unsafe fragments into safe code, as shown in Figure 3. Mutex<T>, a smart pointer in Rust, is designed under the principle of *interior mutability*. *Interior mutability* [44] means that the value bound to a variable can be modified, even if the variable itself is immutable or immutably borrowed. This provides a great functional alternative for accessing or modifying mutable static variables. Meanwhile, the compiler can perform borrow checking for a mutex. When operations violate borrow-checking rules, the code may compile successfully but will suffer from panic at runtime. By replacing mutable static variables with mutexes, Thetis can completely eliminate this type of unsafe fragment, significantly reducing computational overhead. We use LMbench to measure the performance overhead in Section 5.2.

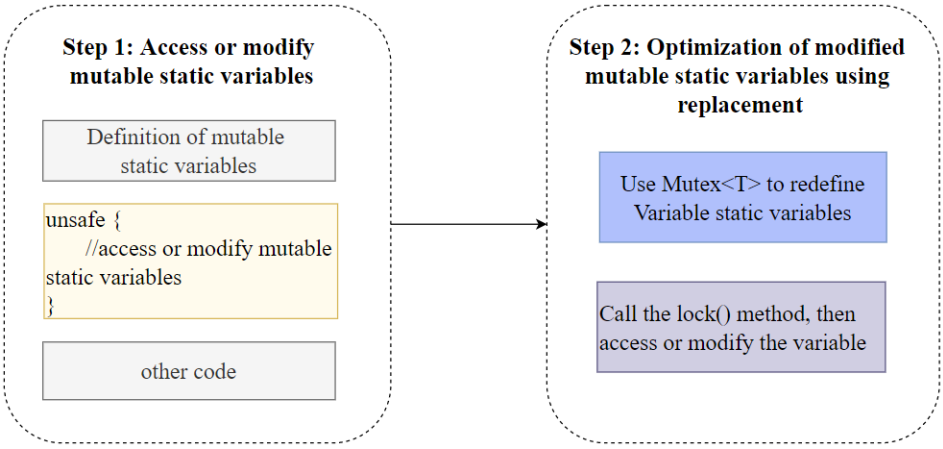

Figure 3. Replacement of mutable static variables.

Mutex replacement can cause deadlock issues. System-level deadlocks occur when multiple processes or threads in a computer system are unable to proceed because each is waiting for a resource that is held by another process. The most common cause of deadlocks is forgetting to use unlock mechanisms later [45]. In the C/C++ language, mutexes must be manually unlocked; otherwise, it will cause deadlocks and reduce system concurrency and performance. Thanks to the lifetime feature of Rust, it will automatically release the lock after the lifetime ends. This greatly reduces the possibility of deadlocks after replacement. In the remaining deadlock situations, we consider that the use of mutable static variables poses a thread safety issue rather than being caused by a mutex. This requires modifications to the code logic. We use an example to verify the correctness of module functionality after replacement. In Listing 5, we provide the code before and after the replacement. Actual operation results show that replacing the module works correctly. Replacement in more complex scenarios is conducted through experiments.

Listing 5. Correctness of mutex replacement function.

```
//Before replacement
static mut TE:u32=0;
fn adt(inc:u32){
    unsafe {TE += inc;}
}
```

```
// After replacement
static TE:Mutex<u32>=Mutex::new(0);
fn adt(inc:u32){
    let t = TE.lock().unwarp();
    *t += inc;
}
```

In general, replacing mutable static variables with mutexes improves code maintainability and clearly expresses the intent and constraints of the code, thereby reducing the possibility of errors.

4.1.2. Replacement of *Union* Types

A *union* is similar to a type *struct* in terms of definition but allows storing different data types in the same memory location. For this reason, when accessing some fields of a *union*, it may overwrite other data's fields or access undefined memory space, resulting in faults. Therefore, in Thetis, the type *struct* is used to replace *union* for optimization.

Thetis optimizes the *union* operations based on feature detection and replacement.

We note that the *struct* syntax is similar in form to the *union* keyword. The *struct* can be used in safe code fragments. Therefore, we propose replacing *union* with *struct* in safety scenarios. However, direct replacement is quite challenging. In this paper, we propose a framework that offers many equivalent alternatives for the syntax.

Firstly, from the perspective of initializing memory languages, in the initialization of *union*, only one of the fields needs to be initialized with a value. The type *struct*, on the other hand, needs to be initialized for all the fields it contains. We achieve this by implementing "#[derive(Default)]" for the replaced struct, where each field is first initialized using the default value before being modified based on the initialized fields in the union. Regarding #[derive (Default)], it automatically uses the default trait code for the struct, as shown in Listing 6, assigning default values to each field.

Listing 6. Default *trait* automatically generated initialization code.

```
#[derive(Default)]
struct Te{f1:u32,f2:f32};
impl Default for Te{
    fn default() -> Self { Self{f1:0, f2:0.0,} }
}
```

Secondly, from the perspective of accessibility, when using the union field multiple times during initialization, the value of each initialization will overwrite the previous memory space. However, struct will retain the results of each assignment. This will result in nonequivalence between the struct and union substitution. To address this issue, we add assertions after replacing the struct fields to prevent access to undefined fields in the union before the replacement, maintaining semantic consistency. To better understand our replacement process, we provide an example, as shown in Listing 7.

Listing 7. Correctness of union replacement function.

```
// After replacement
#[derive[Default]]
struct Te{f1:u32,f2:f32};
let u = Te::default();
let f = u.f1;
assert_eq!(u.f1=1&&u.f2=
       Te.default(),true);
let m = Te::default(); u.f2=2.0;
assert_eq!(u.f1=Te.default()&&u.f2=2.0,true);
```

```
//Before replacement
union Te{f1:u32, f2:f32};
let u = Te{f1:1};
u.f1 = 1;unsafe{let f=u.f1};
let m = Te{f2:2.0};
unsafe {u.f2};
```

Finally, we consider that the replacement of *union* with *struct* directly reduces the use of the "unsafe" keyword while also introducing a small memory overhead. We discuss the overheads in Section 5.2. In general, we analyze the equivalence of replacement frameworks from the perspectives of memory layout and accessibility. Although Listing 7 is a simple example, it also shows that the replaced functionality is correct. In this paper, we did not use formal semantic equivalence proof methods to prove complete equivalence, which is too complex. However, our theoretical and practical analyses of the appeal provide us with reasonable support. The purpose of the design of Thetis is to automatically provide recommended safe code, which can replace unsafe code, with the same functionality.

4.2. Interior Unsafe Encapsulation and ACSL Specification

4.2.1. Measures for Dereferencing Raw Pointers

For dereferencing raw pointers, the correctness of the pointed address cannot be guaranteed because it does not always point to a legal address, and the problem exists that many raw pointers point to the same memory space or NULL. In addition, raw pointers do not have an automatic garbage collection mechanism, which may cause undefined behaviors. Also, Rust does not provide the concept of lifetime for raw pointers, and borrow checking is unavailable. A raw pointer that exceeds its actual lifetime may cause a dangling reference (freeing memory but retaining a pointer to it) issue.

In general, it is safe to create a raw pointer but unsafe to dereference it. The reason for this is that there is no way to guarantee memory safety. To address this problem, Thetis automates the process of locating code corresponding to operations that dereferenced raw pointers to adjust the structure of unsafe fragments through *Interior Unsafe* encapsulation. An encapsulated function is shown in Step 2 in Figure 4.

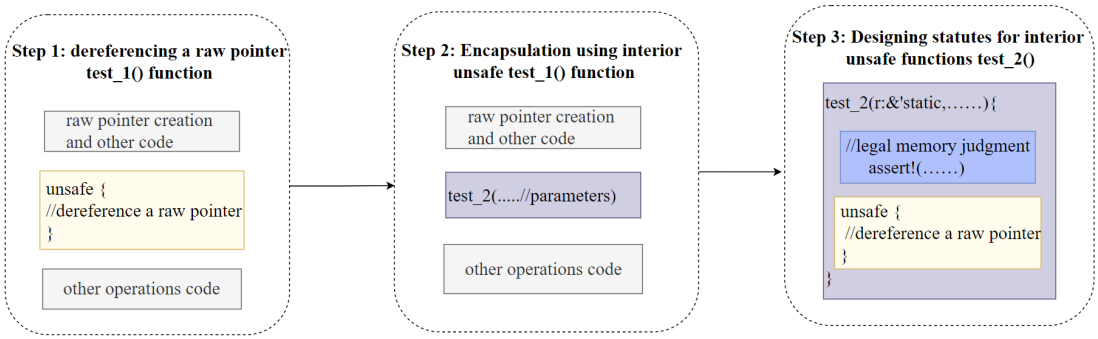

Figure 4. Encapsulation of dereferenced raw pointers.

In addition, we propose a new specification for the encapsulated library functions based on the ACSL method, further reducing the vulnerabilities of the dereferenced raw pointers. The specific design process for the encapsulated functions is as follows:

a. We use *'static* (the standard Rust static lifetime annotation) to explicitly mark the lifetime of raw pointers.
   Although this method may indicate a lifetime exceeding the actual one for a raw pointer, the compiler will be able to perform borrow checking and incorporate the RAII (Resource Acquisition Is Initialization) concept [46] to automatically recycle and clean memory, thus reducing undefined behaviors.

b. We use an assert to constrain the execution state of the program at runtime.
   The goal of an assert in ACSL is to verify the correctness of the code (Figure 5). When the assert code is correct, it can be stated the code that runs before the assert is valid. Thetis inserts the assert before the dereferenced raw pointers, which guarantees that only legal memory can be accessed.

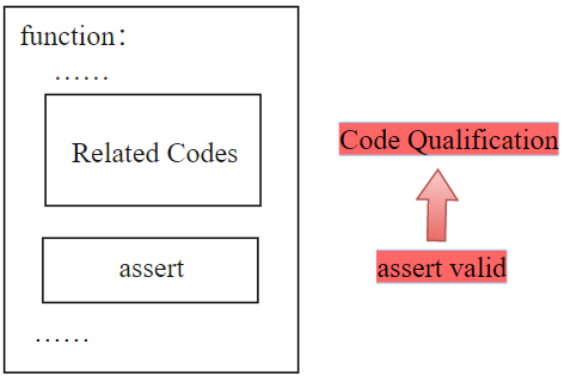

Figure 5. Assert verification.

For optimization, Thetis encapsulates the operation of dereferencing the raw pointer and provides a safe calling interface. Step 3 in Figure 4 demonstrates how to use static analysis to ensure safety after encapsulation. Also, the Thetis function is flexible and extensible. The most common way to safely use raw pointers is to ensure that the dereferenced pointer is not null and is correctly aligned with the reference target type. In our implementation, Thetis provides a more general framework to guide the elimination of unsafe states in unsafe code. Therefore, we use is_null in the assert to provide a minimum level of safety. However, it can be combined with context to incorporate checks on pointer memory addresses, which can further enhance safety. The performance cost of assertion instructions can vary depending on several factors, including the complexity of specific implementations and assertion conditions. We discuss the performance impact in Section 5.2.

4.2.2. Measures for Unsafe Traits/Functions

A *trait* is unsafe when it is implemented using any method with unsafe functions (fns). Therefore, ensuring the safety of the unsafe *trait* is equivalent to reducing the scope of the unsafe functions. Most implementations treat the entire function body of an unsafe function as a large unsafe block, which expands the scope of unsafe fragments, making it more dangerous and difficult to detect unsafe operations.

Based on principles 2 and 3, Thetis takes measures to encapsulate and validate unsafe functions and *trait* code. The method relies on the *Interior Unsafe* concept. It adjusts the scope of the vulnerabilities in functions and then replaces them with an *Interior Unsafe* block, as shown in Step 2 in Figure 6. This method efficiently reduces the scope of the vulnerabilities and makes it easier to check for them. With the scope of the vulnerabilities reduced, the process of improving and verifying safety is further simplified. Additionally, for some *Interior Unsafe* functions, the safety depends on the input and the execution environment [26] so it is essential to describe the context. ACSL provides a function contract design to verify the legitimacy of the code [47,48], which is expressed as a Hoare logic expression (*{pre-condition} function body {post-condition}*), as shown in Figure 7. Thetis annotates function calls with appropriate pre- and post-conditions for *Interior Unsafe* functions according to the verification logic. The assert is used not only to check whether the state provided by the current calling environment is satisfied but also to detect the return status of the function indicated by the post-conditions if the pre-conditions are satisfied. If the function passes the detection of the pre- and post-conditions, the correctness and safety of the scheduled process will be ensured.

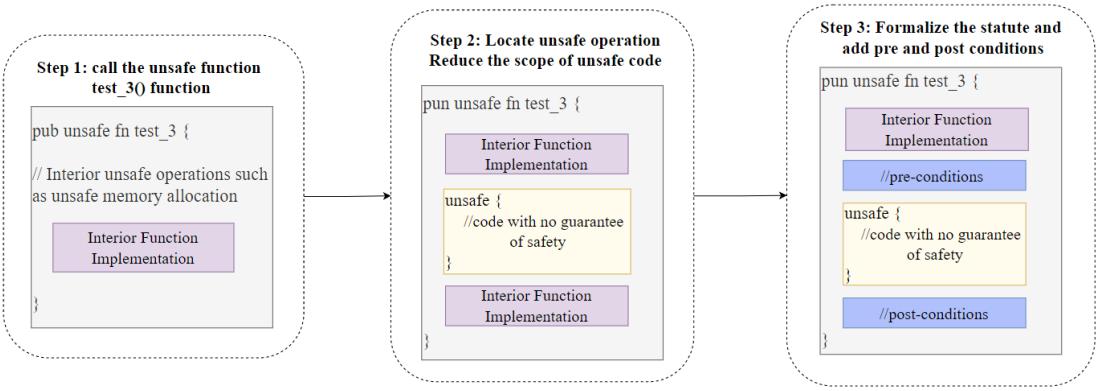

Figure 6. The framework for optimizing unsafe functions or methods.

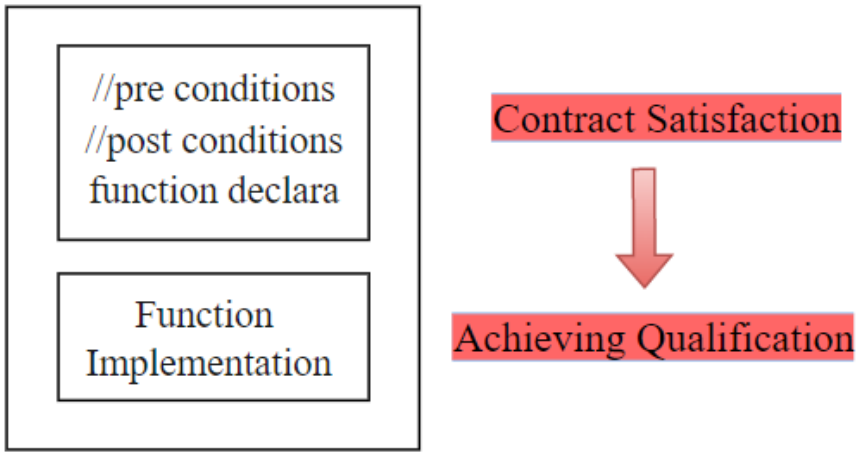

Figure 7. Contract validation functions.

The optimization framework for unsafe functions is illustrated in Figure 6. It is also suitable for unsafe *traits*. When all unsafe functions for a *trait* have been optimized, the "unsafe" keywords can fully be removed from the *trait*.

4.3. Defect Detection Inspection Based on Feature Analysis

Optimizing large-scale systems during development and maintenance is challenging due to the extensive manual effort required [49,50]. To simplify these manual processes and enhance accuracy and speed, our method can automatically detect defects based on feature extraction. It also creates a log of safety checks, which provides a basis for the system to further enhance system safety.

To ensure a strong correlation between the selected features and each category of Unsafe Rust operations, this paper utilizes correlation analysis methods for evaluation. We employ the SPSS data analysis tool to calculate the correlations between features and unsafe operations, which mainly fall into two categories: statistical correlations and probabilistic correlations. For statistical correlations, three calculation methods can be chosen: Pearson [51], Spearman [52], and Kendall [53]. These methods analyze the strength and direction of the relationship between two variables. For probabilistic correlations, Bayesian statistical methods can be used to analyze the degree of association between the two. After calculating the correlation coefficients for the selected features, the operations of each category of Unsafe Rust are categorized based on keywords, function names, and operations, as shown in Table 1.

Table 1. Unsafe Rust feature extraction table.

Type	Keyword	Function Name	Operators
Dereference a raw pointer	*mut/const T	unsafe{. . . }	as or other ways
Call an unsafe function or method	unsafe fn	Self-defined	/
Implement an unsafe trait	unsafe impl	Self-defined	/
Access or modify a mutable static variable	static mut	unsafe{. . . }	read, +, −, *, /. . .
Access field of union	unsafe fn	Self-defined	read, +, −, *, /. . .

The key information is extracted based on the lexical rules of five types of unsafe operations, e.g., keywords, function names, operators, etc., as illustrated in Table 1. The algorithm for feature recognition in defect detection is presented in Algorithm 1. This algorithm takes the operating system source code as input and uses regular expressions to classify, tag, and extract each type of unsafe operation. The output is a report containing specific defect information.

Algorithm 1 Feature-based defect identification

input: Operating system source code files;
Store regular expressions to describe each type of defect in regex_array[5]
for f in fs **do**
  if ends with '.rs **then**
    for i=0; i<5; i++ **do**
      if regex == regex_array[i] **then**
        lst = []
        count[i]++, which is used to count occurrences
        txt[index_1] == {,press into the stack lst.append({)
        According to the matching item left bracket, txt[index_2] == }, out of the stack lst.pop()
        end = index_2 - index_1, which is the defective content, is output to the txt
      else
        Output no match
      end if
    end for
  else
    continue
  end if
end for
output: Defect Report

Compared to the C language, Rust has fewer unsafe operation types, reducing the possibility of false positives and false negatives during large-scale analysis. The method proposed in this paper is based on finite-type feature extraction, which automatically addresses problems. It has the benefit of increasing the accuracy of defect detection and provides guidance for optimizing the system.

5. Experiments and Results

5.1. Safety

One goal of designing and optimizing an operating system is to ensure system safety. When Unsafe Rust is unavoidable in system design, defect detection and elimination methods are important means of enhancing system safety. This section mainly tests the Thetis method proposed in this paper.

Firstly, we selected four open source operating systems written in Rust, namely Redox [10], KatoOS [54], Theseus [55], and BlogOS [56], and analyzed the memory allocation API alloc.rs in the standard library [57].

- Redox v0.6.0 [10]: A Unix-like OS written in Rust that aims to realize the full set of applications and provide system safety for Rust.
- Theseus v1.0 [55]: Uses the Rust compiler to guarantee the safety of the OS, which improves the performance of the system itself.
- KataOS v2.0 [54]: Designed using the Rust language and built on the seL4 [40] kernel, which is mathematically safe, with the features of confidentiality, integrity, and availability.
- BlogOS v2.0 [56]: Used for the teaching of Rust-based OSs and currently includes interrupt handling and memory management functions.

All these OSes are designed using Rust and provide unique safety mechanisms for improving safety compared to traditional language-based systems. However, none of them are specifically optimized for unsafe code fragments and simply use the "unsafe" keyword to isolate unsafe and safe code fragments. We input all systems and libraries into the Thetis defect detection system, and the proportion of each type of unsafe operation was counted, as shown in Table 2.

Table 2. Statistics of unsafe operations.

Name	Dereferencing Raw Pointers	Calling Unsafe Functions	Unsafe Traits	Unions	Mutable Static Variables
Redox [10]	1086	4129	513	454	321
KatoOS [55]	28	368	1	3	10
Theseus [54]	89	335	10	1	12
BlogOS [56]	1	21	6	0	1
ArceOS [58]	94	359	57	25	1
alloc.rs [57]	3	8	1	0	0

We can see that unsafe fragments are unavoidable, even when designing a small standard library. The most commonly used operation is calling unsafe functions or methods, followed by dereferencing raw pointers, even without using union types in some simple systems. Depending on the defect report, Thetis can be used to optimize unsafe operations and reduce memory and thread issues.

Furthermore, to better evaluate the accuracy of feature-based defect detection methods, we assessed them by calculating the detection accuracy (Acc). The calculation of Acc is based on the ratio of correctly predicted samples to all predicted samples as a measure of classification accuracy [59]. The specific calculation is shown in the following formula.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

where TP is the true positive rate, TN is the true negative rate, FP is the false positive rate, and FN is the false negative rate.

We manually inspected the defect detection and classification results of Hubris [60], BlogOS, and ArceOS, and the accuracy values obtained are shown in Table 3. From these results, it can be observed that due to the presence of behaviors such as dereferencing raw pointers through library encapsulated functions, dereferencing raw pointers in "unsafe fn", and accessing mutable static variables in the system design, there were cases of duplicate classifications and misclassifications in the identification process. However, on average, the proposed defect detection method in this paper achieved an average detection accuracy of 84.6%. We selected Miri, a commonly used defect-checking method in Rust, to perform defect detection on ArceOS and calculate the accuracy of this method. As shown in Table 3, it achieved an accuracy of around 63.6%, and there were numerous cases of false negatives and false positives in the detection results. The accuracy of our method was superior to that of Miri by approximately 20%.

Table 3. Defect detection accuracy.

System	Detection Method	Detection Accuracy
Hubris	Defect detection based on feature extraction	87.8%
BlogOS	Defect detection based on feature extraction	80.6%
ArceOS	Defect detection based on feature extraction	85.4%
ArceOS	Miri (Defect Detection Techniques for Rust Language)	63.6%

To demonstrate the flexibility of Thetis, we employed it to fix BlogOS and ArceOS. Considering the overhead of manually writing assertions and setting pre- and post-conditions, the method proposed in this paper mainly improved the axfs, axlog, and axsnyc modules in the ArceOS modular operating system.

According to the defect report, we found defects in memory.rs in BlogOS, such as the active_level_4_table() function shown in Listing 1 (in Section 2). This function was declared unsafe, and its vulnerability mainly derived from two aspects: (i) the correctness of "physical_memory_offset", which should map physical memory to the correct virtual memory; and (ii) the memory safety of dereferencing a raw pointer. Relying on Thetis, we first used the *Interior Unsafe* concept to encapsulate the unsafe operation of dereferencing a raw pointer. This reduced the range of unsafe fragments. Through static analysis, we added pre-conditions to guarantee the correctness of "physical_memory_offset". The repaired code is shown in Listing 8. The optimization process of unsafe *traits* is similar to that of unsafe functions. First, the specific causes of the unsafe trait are analyzed. Then, it is encapsulated and verified using the *Interior Unsafe* and ACSL methods. For unsafe traits that cannot be encapsulated, pre- and post-conditions are added for each of the methods. Thus, developers can be further guided in performing the subsequent checking process.

Listing 8. Optimize active_level_4_table().

```
fn active_level_4_table(physcial_memory_offset:VirtAddr)->
&'static mut PageTable {
    let (level_4_table_frame, _) = Cr3::read();
    let phys = level_4_table_frame. start_address();
    //Determine if the passed physcial_memory_offset is
        correct
    assert!(physcial_memory_offset, VirtAddr::new(boot_info
        . physical_memory_offset);
    let virt = physcial_memory_offset + phys. as_u64();
     let page_table_ptr:*mut PageTable = virt.as_mut_ptr();
     //Encapsulation of dereference a  raw pointers
     raw_usage(page_table_ptr);
}
pub fn raw_usage(r:&'static mut PageTable) -> &'static mut
     PageTable {
     assert!(!r. is_null());
     unsafe { &mut *r }
}
```

The unsafe operation of accessing and modifying static variables was found in the source gdt.rs according to the defect report, as shown in Listing 3 (in Section 2)). Under the guidance of Thetis, we replaced it with a mutex, and the optimized result is shown in Listing 9.

Listing 9. Accessing static variable optimization process.

```
let STACK = Mutex::new(vec![0; STACK_SIZE]);
let stack_guard = STACK.lock().unwrap();
let stack_start =VirtAddr::from_ptr(stack_guard.as_ptr());
```

Since a union is not used in BlogOS and ArcoOS, the processing is not described here. We checked the repaired system again and compared it to the original one. The number of unsafe operations changed, as shown in Figure 8.

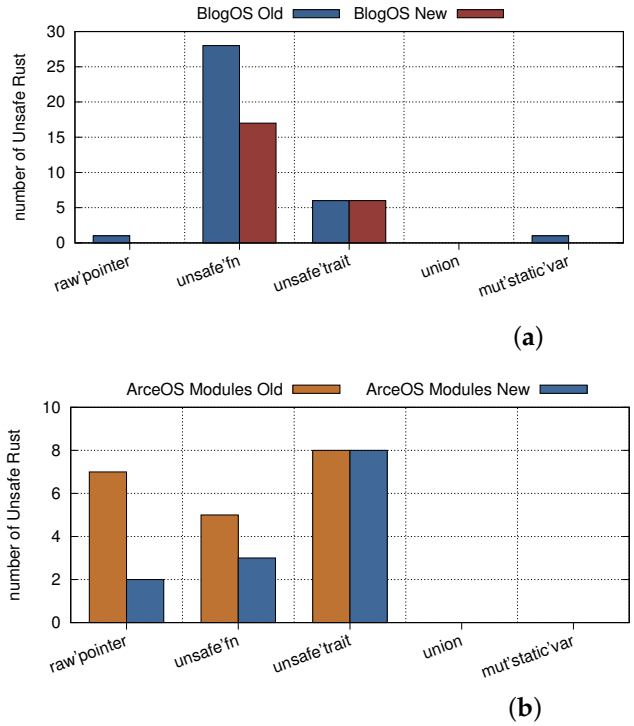

Figure 8. Comparison chart of unsafe operations. (**a**) Comparison of BlogOS before and after repair. (**b**) Comparison of three modules selected from ArceOS before and after repair.

In Figure 8, it can be seen that after using Thetis to repair the BlogOS and ArceOS modules, the overall number of unsafe operations decreased by about 41.3% and 45%, respectively. The unsafe operations of dereferencing raw pointers and accessing/modifying mutable static variables were completely eliminated through replacement and encapsulation, which effectively and directly reduced unsafe usage. For the unsafe functions/*traits*, the scope and state of vulnerabilities were reduced through encapsulation and static analysis. In addition, some unsafe functions could not be directly replaced with safe ones. With the scope of the vulnerabilities reduced, the state complexity of formal verification was also reduced. Therefore, the potential for designing and verifying truly safe systems was increased. Here, we should note an additional problem. A standard/core library is widely used in Rust-based OSs, and this library almost always contains unsafe fragments. Therefore, Thetis should also be used to optimize this library along with the project development.

Finally, to verify the effectiveness of Thetis in defect reduction, we used Miri to detect undefined behaviors in ArceOS modules before and after repair. The detection results show that the potential for UBs in the system decreased by approximately 50%. The Thetis method proposed in this article significantly reduced Unsafe Rust operations and minimized interactions between Unsafe Rust and Safe Rust. Specifically, as shown in Listing 10 after eliminating the issues, ArceOS was tested again with Miri, and the direct use of Unsafe Rust operations (such as dereferencing raw pointers in the table) in the system was eliminated, thereby reducing the use of Unsafe Rust. Through analysis combined with inspecting the source code, operations like "run_queue::init()" and "INIT.call_once(thread::init_scheduler)", as shown in Listing 10, contained vulnerabilities resulting from the interaction between directly calling Unsafe Rust (as indicated by dereferencing raw pointers in the table) and Safe Rust code. The proposed method improved the safety of Unsafe Rust, reduced interactions, and limited the scope of vulnerabilities.

Listing 10. Instructionsfor each type of Unsafe Rust operation.

```
self.0 as *mut u8  //Unsafe Rust: dereference raw pointers
//The safety risks brought by the interaction between
   Unsafe Rust and Safe Rust
crate::run\_queue::init();
INIT.call\_once(thread::init\_scheduler);
```

5.2. Performance

Another goal in the design and optimization of an operating system is high performance. Unsafe Rust, when used with the "unsafe" keyword, bypasses the compiler's checks, which can lead to performance improvements to some extent. Thetis, which is based on minimizing the use of Unsafe Rust, incurs certain performance overheads. This section mainly focuses on the analysis and testing of Thetis's performance.

For defect detection, the feature-based automated defect detection method proposed in this paper utilizes static feature extraction and regular expression matching. As shown in Table 4, compared to traditional static detection methods, our method is based on source code analysis, without the need for actual program execution. Dynamic detection methods, on the other hand, require program execution and monitoring, which incur significant time and resource costs. However, dynamic detection can cover more execution paths and potential error scenarios through monitoring, whereas static analysis may have a higher risk of false negatives and false positives. Our method, which is based on limited feature extraction, narrows down the detection scope, reduces the possibility of false negatives and false positives, and does not incur significant overheads. Lastly, considering deployment and usability aspects, dynamic detection methods usually require deployment and program execution in real environments, requiring certain technical expertise and resources, and in some cases, they may impact the program's performance. In contrast, the feature extraction and static detection method proposed in this paper can be directly applied to the source code or compiled code, making it relatively straightforward to use. Therefore, the automated defect detection method proposed in this paper improves detection accuracy without causing excessive performance overhead.

Table 4. Performance analysis of defect detection methods.

Defect Detection Method	Time and Resources	Coverage	Deployment Difficulty
Feature extraction	Source level	All unsafe code	Automated, Simple
Static method	Source code/compiler	Static structure/logic	Relatively simple
Dynamic method	Runtime monitoring	Runtime behavior	Requires technology/resources

For defect elimination, firstly, a theoretical analysis of the potential overhead in defect elimination methods is performed: (1) For unsafe operations accessing union fields, this paper uses struct as an alternative. During this process, it is necessary to allocate memory space and assign values to all fields in the union, which incurs certain memory overheads; (2) For unsafe operations accessing/modifying static variables, this paper uses mutex locks as an alternative, as the use of locks can increase system runtime overhead. When there is no conflict, the processor cost of acquiring and releasing locks is equivalent to the cost of CAS instructions. If there is a conflict, it will enter a sleep state waiting for wake-up, which increases the cost of context switching and thread scheduling. (3) For other types of unsafe operations, interior unsafe encapsulation may incur overheads.

Despite the performance overhead of the method proposed in this paper, combining the defect detection results reveals that unions and mutable static variables account for a small proportion of system design, and the switching overhead caused by internal unsafe encapsulation is theoretically small as well. Therefore, theoretically, the defect elimination

method designed in this paper has little impact on the performance of Rust systems. In order to accurately measure these overheads, this section utilizes LMbench [61] to test the Rust system (specifically, the rCore system [62]) before and after defect elimination. The test results, as shown in Figure 9, demonstrate that the overall performance decreases by approximately 1.076%, maintaining the high-performance characteristics of the Rust system.

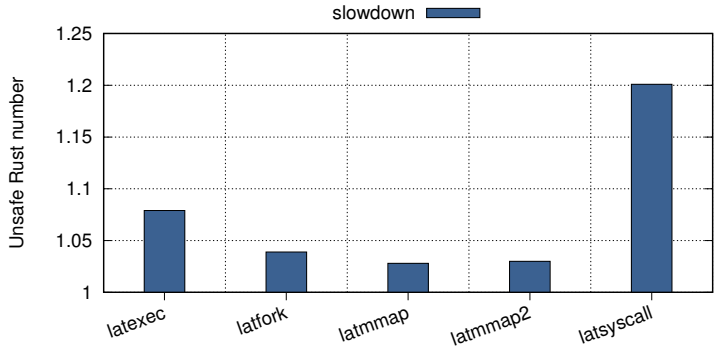

Figure 9. LMbench performance.

6. Conclusions and Future Work

In this paper, we propose Thetis, a novel methodology that can automatically provide suggested safe code, thereby minimizing unsafe code fragments and improving system safety. Essentially, Thetis combines replacement and encapsulation for *Interior Unsafe* functions, effectively reducing the scope of unsafe fragments and the interactions between unsafe and safe code. Additionally, the reduction in the scope of unsafe code simplifies formal verification, offering greater potential for designing safer OSs. Furthermore, new ACSL formal statutes are applied to reduce the unsafe potential of the encapsulated *Interior Unsafe* functions. Thus, raw pointers can achieve lifetime support, and unsafe functions can be partially verified by condition limitations. Moreover, automatic defect detection and optimization based on feature extraction are introduced, which reduces the complexity of manual checking and improves development efficiency.

Overall, Thetis is flexible and extensible. It allows developers to expand the scope of safety checks and further enhance system safety based on context and experience. And, it effectively improves development efficiency. With the development of the Rust programming language, there are an increasing number of studies that focus on enhancing Rust's safety. These can be easily integrated into encapsulated library functions. Benefiting from Thetis's extensibility, developers can minimize source modifications. Thetis can not only directly reduce unsafe operations in the OS source code but also reduce the occurrence of false negatives and false positives in program testing, thereby greatly improving system safety and reliability. In the future, Thetis can also be combined with machine learning technologies to further enhance system safety optimization and verification.

Author Contributions: Conceptualization, R.J., P.D., Y.D., R.W. and Z.J.; methodology, R.J., P.D., Y.D., R.W. and Z.J.; software, R.J., P.D. and Z.J.; validation, R.J., P.D. and Z.J.; writing—original draft preparation, R.J., P.D., Y.D., R.W. and Z.J.; writing—review and editing, R.J., P.D., Y.D., R.W. and Z.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China grant numbers u19a2060 and 62172431, and the Pre-Research Project grant number 31511100101.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author. The data are not publicly available due to commercial restrictions.

Acknowledgments: The authors would like to thank the anonymous reviewers for their comments and feedback.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gens, D.; Schmitt, S.; Davi, L.; Sadeghi, A.-R. K-Miner: Uncovering Memory Corruption in Linux. In Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, CA, USA, 18–21 February 2018.
2. Wu, S.; Wang, K.; Jin, H. Research Situation and Prospects of Operating System Virtualization. *J. Comput. Res. Dev.* **2019**, *56*, 58–68.
3. Concurrency in Operating System. 2019. Available online: https://www.geeksforgeeks.org/concurrency-in-operating-system/ (accessed on 30 May 2019).
4. Memory Management. Available online: https://www.tutorialspoint.com/operating_system/os_memory_management.htm (accessed on 13 November 2023).
5. Nagarakatte, S.G. *Practical Low-Overhead Enforcement of Memory Safety for C Programs*; University of Pennsylvania: Philadelphia, PA, USA, 2012.
6. Stack Overflow. Stack Overflow Developer Survey 2020. 2020. Available online: https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted (accessed on 1 January 2020).
7. Stack Overflow. Stack Overflow Developer Survey 2021. 2021. Available online: https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted (accessed on 1 January 2021).
8. Rust Ownership. Available online: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html (accessed on 3 May 2019).
9. Rust Lifetime. Available online: https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html (accessed on 3 May 2019).
10. Redox. The Redox Operating System [CP/OL]. Available online: https://www.redox-os.org (accessed on 1 January 2015).
11. Servo. The Servo Browser Engine [CP/OL]. Available online: https://servo.org (accessed on 1 January 2016).
12. Matsakis, N.D.; Klock, F.S., II. The rust language. In Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology, Portland, NY, USA, 18–21 October 2014; pp. 103–104.
13. Rustsec. Rustsec Advisory Database. 2021. Available online: https://github.com/RustSec/advisory-db (accessed on 7 July 2021).
14. The Rust Programming Language. Available online: https://rustwiki.org/zh-CN/book/ch19-01-unsafe-rust.html (accessed on 9 on February 2023).
15. Zhu, S.; Zhang, Z.; Qin, B.; Xiong, A.; Song, L. Learning and Programming Challenges of Rust: A Mixed-Methods Study. In Proceedings of the ICSE'22: 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 21–29 May 2022.
16. Rust-Book. Available online: https://www.rust-lang.org/ (accessed on 3 May 2019).
17. Formal_Verification. Available online: https://en.wikipedia.org/wiki/Formal_verification (accessed on 6 May 2007).
18. Bu, L.; Xie, D.-B. Formal Verification of Hybrid System. *J. Softw.* **2014**, *2*, 219–233.
19. Demri, S.; Laroussinie, F.; Schnoebelen, P. A parametric analysis of the state-explosion problem in model checking. *J. Comput. Syst. Sci.* **2006**, *72*, 547–575. [CrossRef]
20. Wang, J.; Zhan, N.-J.; Feng, X.-Y.; Liu, Z.-M. *Overview of Formal Methods*; Carnegie Mellon University: Pittsburgh, PA, USA, 1998.
21. Zhang, G.; Wang, P.-F.; Yue, T.; Zhou, X.; Lu, K. MEBS: Uncovering Memory Life-Cycle Bugs in Operating System Kernels. *J. Comput. Sci. Technol.* **2021**, *36*, 1248–1268. [CrossRef]
22. Undefined Behavior. Available online: https://en.wikipedia.org/wiki/Undefined_behavior (accessed on 10 January 2006).
23. C++ Undefined Behaviors. Available online: https://en.cppreference.com/w/cpp/language/ub (accessed on 8 November 2023).
24. A Guide to Undefined Behavior in C and C++. Available online: https://blog.regehr.org/archives/213 (accessed on 1 January 2019).
25. Qin, B.; Chen, Y.; Yu, Z.; Song, L.; Zhang, Y. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20), London, UK, 15–20 June 2020.
26. Qin, B. *An Empirical Study on the Safety of Real-World Rust Programs*; Beijing University of Posts and Telecommunications: Beijing, China, 2021.
27. Jung, R.; Jourdan, J.; Krebbers, R.; Dreyer, D. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* **2018**, *2*, 1–34. [CrossRef]
28. Astrauskas, V.; Bíly, A.; Fiala, J.; Grannan, Z.; Matheja, C.; Müller, P.; Poli, F.; Summers, A.J. The Prusti Project: Formal Verification for Rust. *NASA Form. Methods* **2022**, *13260*, 88–108.
29. Astrauskas, V.; Müller, P.; Poli, F.; Summers, A.J. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* **2019**, *3*, 147. [CrossRef]
30. Liang, H.; Pei, X.; Jia, X.; Shen, W.; Zhang, J. Fuzzing: State of the art. *IEEE Trans. Reliab.* **2018**, *67*, 1199–1218. [CrossRef]
31. Jiang, J.; Xu, H.; Zhou, Y. RULF: Rust Library Fuzzing via API Dependency Graph Traversal. In Proceedings of the ASE'21: 36th IEEE/ACM International Conference on Automated Software Engineering, Melbourne, Australia, 15–19 November 2021.
32. Luo, J.; Liu, M.; Luo, Y.; Chen, Z.; Zhang, Y. A Runtime Monitoring Based Fuzzing Framework for Temporal Properties. In Proceedings of the 32nd IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW, Wuhan, China, 25–28 October 2021.

33. Cadar, C.; Sen, K. Symbolic execution for software testing: Three decades later. *Commun. ACM* **2013**, *56*, 82–90. [CrossRef]
34. Rustybox. Available online: https://github.com/samuela/rustybox (accessed on 17 January 2019).
35. Chen, Z.; Chen, Z.; Shuai, Z.; Zhang, Y.; Pan, W. Synthesizing smart solving strategy for symbolic execution. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Virtual, 21–25 December 2020.
36. Bae, Y.; Kim, Y.; Askar, A.; Lim, J.; Kim, T. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In Proceedings of the SOSP'21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Koblenz, Germany, 26–29 October 2021.
37. Li, Z.; Wang, J.; Sun, M.; Lui, J.C.S. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In Proceedings of the CCS'21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual, 15–19 November 2021.
38. Cui, M.; Chen, C.; Xu, H.; Zhou, Y. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *ACM Trans. Softw. Eng. Methodol.* **2022**, *32*, 1–21. [CrossRef]
39. The Rust Standard Library. Available online: https://doc.rust-lang.org/std/sync/struct.Mutex.html (accessed on 3 May 2019).
40. seL4. Available online: https://sel4.systems/ (accessed on 5 June 2009).
41. Miri. Available online: https://github.com/rust-lang/miri (accessed on 22 May 2019).
42. Péter, R.; Rice, H.G. Classes of recursively enumerable sets and their decision problems. *Trans. Am. Math. Soc.* **1953**, *74*, 358–366; Erratum in *J. Symb. Log.* **1954**, *19*, 121–122.
43. Chess, B.; McGraw, G. Static analysis for security. *IEEE Secur. Priv.* **2004**, *2*, 76–79. [CrossRef]
44. Interior Mutability. Available online: http://rcore-os.cn/rCore-Tutorial-Book-v3/chapter2/3batch-system.html#term-interior-mutability (accessed on 15 June 2019).
45. Coffman, E.G.; Elphick, M.J.; Shoshani, A. System deadlocks. *Comput. Surv. (CSUR)* **1971**, *3*, 67–78. [CrossRef]
46. RAII. Available online: https://zh.m.wikipedia.org/zh-hans/RAII (accessed on 12 January 2012).
47. Kirchner, F.; Kosmatov, N.; Prevosto, V.; Signoles, J.; Yakobowski, B. Frama-C: A software analysis perspective. *Form. Asp. Comput.* **2015**, *27*, 573–609. [CrossRef]
48. ACSL and Frama-C Validation Tools. Available online: https://zhuanlan.zhihu.com/p/266701781 (accessed on 19 October 2020).
49. Kai, Z.; Yi, R.; Zhe, W.; Guan, J.-B.; Fang, Z.; Zhao, Y.-K. Classification and Analysis of Ubuntu Bug Reports Based on Topic Model. *Comput. Sci.* **2020**, *47*, 35–41.
50. Li, Y.; Huang, C.-L.; Wang, Z.-F.; Yuan, L.; Wang, X.-C. Survey of Software Vulnerability Mining Methods Based on Machine Learning. *J. Softw.* **2020**, *31*, 2040–2061.
51. Cohen, I.; Huang, Y.; Chen, J.; Benesty, J.; Benesty, J.; Chen, J.; Cohen, I. Pearson correlation coefficient. *Noise Reduct. Speech Process.* **2009**, 1–4.
52. Myers, L.; Sirois, M.J. Spearman correlation coefficients, differences between. *Encycl. Stat. Sci.* **2004**, *12*. [CrossRef]
53. McLeod, A.I. Kendall rank correlation and Mann-Kendall trend test. *R Package Kendall* **2005**, *602*, 1–10.
54. KataOS. Available online: https://opensource.googleblog.com/2022/10/announcing-kataos-and-sparrow.html (accessed on 10 May 2022).
55. Alexander, B.K. *Theseus: Rethinking Operating Systems Structure and State Management*; Rice University: Houston, TX, USA, 2020.
56. Blog_os. Available online: https://github.com/phil-opp/blog_os (accessed on 7 June 2018).
57. Memory Allocation APIs. Available online: https://doc.rust-lang.org/src/std/alloc.rs.html#1-416 (accessed on 3 May 2019).
58. ArceOS. Available online: https://github.com/rcore-os/arceos (accessed on 1 June 2022).
59. Veropoulos, K.; Campbell, C.; Cristianini, N. Controlling the sensitivity of support vector machines. *Proc. Int. Jt. Conf.* **1999**, *55*, 60.
60. Biffle, C.L. On Hubris and Humility: Developing an OS for Robustness in Rust. In Proceedings of the Open Source Firmware Conerence 2021, Virtual, 21 September 2021.
61. McVoy, L.W.; Staelin, C. lmbench: Portable Tools for Performance Analysis. In Proceedings of the USENIX Annual Technical Conference, San Diego, CA, USA, 22–26 January 1996; pp. 279–294.
62. rCore. Available online: https://github.com/rcore-os/rCore-Tutorial-v3 (accessed on 15 June 2019).