# Simulator of Theseus: Substituting parts for a Memory Safe Simulator

**Jonathan Mitchell**
**Thales Training & Simulation**
**Melbourne, VIC, Australia**
**jonathan.mitchell@thalesgroup.com.au**

**Kerryn Spanhel**
**Thales Training & Simulation**
**Rydalmere, NSW, Australia**
**kerryn.spanhel@thalesgroup.com.au**

## ABSTRACT

The Ship of Theseus is a thought experiment that asks the question “if a ship has had all its components replaced one by one, would it remain the same ship?”. If we did the same to a simulator, replacing its software components one by one with a memory safe implementation, would it remain the same simulator? We assert the answer is yes, but with the attribute of being more cyber resilient.

Cyber threats are increasing due to a deterioration in the global security environment and increasing sophistication of cyber criminals. At the same time, the need to interconnect simulations to enable collective training or training services is growing. This leads to an increased need for cyber resilience of simulators. Impacts to the availability or integrity of these simulators can reduce the availability or competency of trained professionals, risking disruptions to important elements of the economy.

Simulators contain complex software that carries risk in the form of potential software vulnerabilities. Memory safety vulnerabilities are a prevalent type of software vulnerability that threat actors routinely exploit. These vulnerabilities pose such a risk that a joint statement was made by the Five Eyes National Security Agencies calling for the transition to Memory Safe Languages (MSLs) in Software Systems. This paper explores the utility and benefit of MSLs in the context of simulation and provides an approach to iteratively replace small legacy software components of a simulator written in C/C++ using Rust. This approach leverages Rust’s ability to generate C-bindings to integrate with the legacy system. A proof-of-concept of this approach is presented with the lessons learnt and general advice on how to transition legacy simulation software components to an MSL, eliminating this class of software vulnerability through the memory safety guarantees provided by Rust.

## ABOUT THE AUTHORS

**Jonathan Mitchell** is a Simulation Architecture and Modelling Specialist with over a decade of experience with Thales Training & Simulation. A multi-disciplined engineer with strengths in Systems Engineering and Mechatronics, Jonathan is passionate about innovation and the exploration of novel technologies and methodologies for simulation. Jonathan has worked extensively in Flight Simulation for both civil and military end users. Jonathan is a Chartered Professional Engineer with Engineers Australia and holds Bachelors of Computer Science and Engineering and a Master of Philosophy in the field of non-linear state estimation. Jonathan’s areas of professional interest are Simulation Architecture, Simulation Modelling, Cloud Native Technologies, Cyberworthiness, DevSecOps and Risk Management.

**Kerryn Spanhel** is a Senior Systems/Software Engineer with over six years’ experience in Training and Simulation at Thales. Kerryn is a Chartered Professional Engineer with Engineers Australia and holds a Bachelor of Engineering in Mechanical and Mechatronic Engineering, and a Diploma of Engineering Practice. Kerryn has led complex engineering change proposals on in-service simulators, working across the full systems engineering lifecycle, from requirements to verification, validation, and acceptance. Kerryn is passionate about creating innovative solutions within budget and schedule, delivering direct positive training outcomes to the end user, and a maintainable solution for the support team.

# Simulator of Theseus: Substituting parts for a Memory Safe Simulator

**Jonathan Mitchell**
**Thales Training & Simulation**
**Melbourne, VIC, Australia**
**jonathan.mitchell@thalesgroup.com.au**

**Kerryn Spanhel**
**Thales Training & Simulation**
**Rydalmere, NSW, Australia**
**kerryn.spanhel@thalesgroup.com.au**

## INTRODUCTION

Simulators are critical assets that support the training of skilled professionals operating in high-consequence industries, such as medical, rail, and defence. These industries rely on highly trained professionals to ensure the function of vital services to society. As such, these simulators are high-value cyber targets due to this criticality. The data they process and store also contains immense value to adverse actors. Successful cyber attacks on simulator assets can cause grave damage to the economy or national interests. For instance, the Aviation and Aerospace sectors are facing growing cyber threats and increases in realised cyber attacks, where their critical infrastructure label make them enticing targets to Advanced-Persistent Threat (APT) groups and hacktivist collectives (Resecurity, 2024).

With the growing sophistication in cyber threats and use of Artificial Intelligence lowering the skill barrier to cybercrime (Burton, Janjeva, Moseley, & Alice, 2025), the need for cyber resilience in simulators is paramount. This is exacerbated by the rise of cloud computing and internet facing systems, exposing new cyber attack surfaces for contemporary simulator and training centre architectures.

Simulators now require the ability to withstand, respond to, and recover from cyber attacks, while continuing to operate with minimal disruption. This cyber resilience is essential to ensure safety, security, high assurance and high availability to deliver the training outcomes required to maintain critical industries.

To improve cyber resilience of simulators it is important to minimise vulnerabilities in developed software. Studies have shown that 70% of software vulnerabilities are due to memory safety issues (MSRC Team, 2019). This is a disproportionate attribution of software vulnerabilities for a small subset of software vulnerability types. Memory safety vulnerabilities are software weaknesses such as buffer overflows, use-after-free and out-of-bounds memory access (MITRE, 2025). Because memory safety issues are disproportionately responsible for the majority of software vulnerabilities in software systems, a joint statement was made by the Five Eyes National Security Agencies calling for the transition to Memory Safe Languages (MSLs) in Software Systems (Cybersecurity and Infrastructure Security Agency, 2023).

These types of vulnerability are often the entry points for malicious actors seeking to compromise systems. Using contemporary memory safe languages, such as Rust, it is possible to eliminate the entire class of memory safety related vulnerabilities at compile time. This provides a significant reduction, up to 70%, of software vulnerabilities in developed software. Eliminating memory safety defects in simulators provides significant improvements in cyber resilience, resulting in systems that are much harder to exploit.

Memory safety in software can be approached through three typical means. The first is to use Static Analysis Security Testing (SAST) and Dynamic Analysis Security Testing (DAST). The next two approaches to memory safety are by design choices in the programming language itself, specifically Garbage Collection and Memory Ownership. Examples of languages that use these various approaches to memory safety is given in Figure 1.

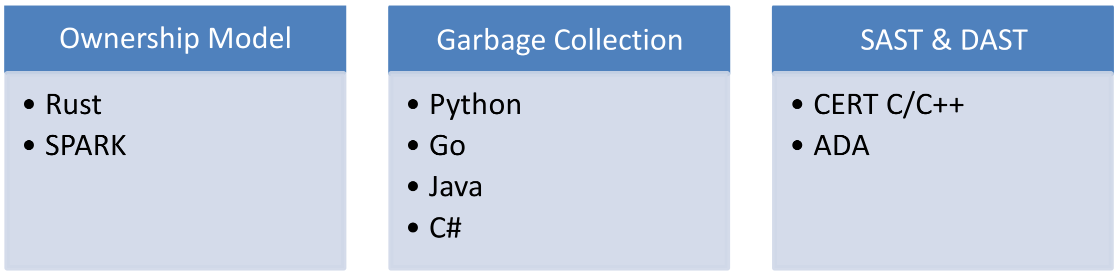


**Figure 1: Memory Safety Approaches**

SAST and DAST tools rely on analysing source code using rules and instrumenting compiled code with detectors to detect undefined behaviour in software. Examples of SAST tools are Coverity and Sonarqube. Examples of DAST tools are fuzztest, valgrind, AddressSanitizer, and MemorySanitizer. A SAST/DAST approach to memory safety is typically utilised by C/C++ applications as they inherently lack memory safety in the design of the language.

Garbage Collection provides automatic memory management, releasing the programmer from having responsibility of the allocation and deallocation of memory. This prevents memory leaks and provides efficient use of memory, eliminating memory management errors such as use-after-free. Garbage collectors also typically enforce bounds checks on memory allocations, eliminating buffer overflows and out-of-bounds memory access errors. Languages that employ garbage collection typically are not appropriate for latency sensitive applications due to the lack of control over the garbage collector. This lack of control can lead to the garbage collector automatically cleaning up memory at an inopportune time during execution and causing latency sensitive processes to miss their performance deadlines.

The Memory Ownership model was pioneered by Rust and is a novel approach to memory safety. Memory Ownership is a set of rules that are enforced by the compiler, that if any are violated, the program won't compile. It works on three fundamentals principles; each allocation of memory (stack or heap) has an owner, there can only be one owner at a time, when the owner goes out of scope the memory will be unallocated. These simple rules enforced by the compiler ensure memory safety without any runtime overhead.

In this paper we advocate for adopting Rust in the development of simulation software. Rust is a performant language suitable for latency sensitive applications that eliminates memory safety vulnerabilities through its use of a Memory Ownership model which significantly reduces software vulnerabilities overall and greatly improves the cyber resilience of simulators.

## The Case for Rust in Simulation

Rust is a memory safe language that is appealing for simulation use cases due to its zero cost abstractions, like C++, and its suitability for latency sensitive and performance critical software. Simulators involve complicated algorithms, mathematics, data structures and concurrency. This software complexity invites human error during software development that leads to defects in the software. These defects can be challenging to detect at runtime and become long lived. Such defects could be exploitable by adversaries and make Simulators vulnerable to cyber-attack. It is critical that software defects, in particular memory-safety defects are eliminated or minimised to improve the cyber resilience of simulators. Rust provides the means for eliminating this class of software defect.

Rust achieves its memory safety through its unique ownership model and borrow checker ensuring that memory is correctly managed at compilation, eliminating memory safety issues without any runtime overhead (Klabnik & Nichols, 2023). The language also has numerous features in addition to the ownership model and borrow checker that promote safer and less vulnerable software.

The Rust programming language is supported by an integrated out of the box tooling, including a document generator, package manager, code formatter, unit testing and syntax linter. Rust not only results in safer software but provides a productive standardised environment for developers without the need for third-party tools typical of C/C++ development.

Rust's strict type system and compiler checks provides the advantage of ensuring code correctness at compilation. This significantly reduces the discovery of defects at runtime where it can be challenging to investigate and determine the root cause, particularly with race conditions and memory safety issues. By identifying issues at compilation, this considerably reduces the likelihood of defects in software.

Rust eliminates the concept of undefined behaviour (which can be prevalent in C/C++) in the safe subset of the language through the checks provided by the Rust compiler and its memory ownership model. It is possible in Rust to opt out of its safety guarantees using the *unsafe* keyword. This is an important capability for when writing performance critical algorithms or writing drivers that interface at a very low level. The benefit of explicit use of the *unsafe* keyword ensures it is easy to distinguish between safe and unsafe code, limiting the scope of potentially vulnerable code, and allowing for focused auditing to ensure the invariants of the unsafe code are maintained. The Rust toolchain also comes equipped with an undefined behaviour detection tool, Miri, that provides detection of unsafe code that fails to uphold its safety requirements.

Rust's type system further strengthens software security by ensuring that only valid operations are allowed on valid data types, minimizing logic errors that could introduce vulnerabilities. Enforcing strict rules on data types and their use reduces errors as problems are identified at compile time and improves code maintainability. It also enhances collaboration in a code base, making it easier for developers to understand code's intent and behaviour. This is critically important for Application Programming Interfaces (APIs) where third parties may be integrating with software systems. Rust's type system allows for well-defined APIs that disallow incorrect use, making them easier and safer to use.

Additionally, Rust has several features for error handling. Unhandled or unrecoverable errors cause the application to enter a *panic* state, which results in the application printing a failure message, cleaning up after itself and ceasing execution. This is desirable as unhandled errors that allow for software to continue executing results in undefined behaviour and can result in software vulnerabilities. Recoverable errors can be handled through Rust's *Result* type. This allows for a developer to encode detailed information on the errors and conditions that can be expected during operation, providing comprehensive handling of all anticipated fault scenarios. This not only improves the robustness of software but further reduces the likelihood of unhandled errors, providing a vector of attack by adversarial actors.

Concurrency is another area where Rust provides guarantees on code correctness. Rust leverages its memory safety and type system to analyse correctness of many concurrency issues at compile time (Klabnik & Nichols, 2023). This means many concurrency errors are detected during compilation rather than runtime ensuring that software is free of subtle bugs and undefined behaviour due to race conditions. In contrast, other languages such as C++ require manual management of threads and synchronisation, making it error-prone and more challenging to implement safe concurrent systems.

## Rust in Industry

The adoption of Rust in industry has been rapidly increasing over the past few years, particularly in areas where performance, safety, and security are paramount. Rust's focus on memory safety without sacrificing performance makes it particularly appealing to industries like simulation where timing constraints are important, such as flight simulators. Rust, although a relatively young programming language, is mature and suitable for production systems.

Microsoft's adoption of Rust has received significant attention in recent years, with its focus on improving the security and reliability of their software. Microsoft has made significant steps towards supporting the use of Rust in Windows, and directly using Rust as part of Windows. Microsoft have developed the windows and windows-sys Rust crates to allow Rust developers to call any Windows API (Microsoft, 2025). David Weston, Director of OS Security at Microsoft, announced at Microsoft's BlueHat IL 2023 conference that parts of the Windows kernel will be rewritten in Rust to improve security (Weston, 2023).

Rust has also been introduced to the Linux Kernel, officially being introduced in version 6.1 (Torvalds, 2022). The objective for incorporating Rust was to achieve similar goals to those outlined in this paper (Ojeda , 2021). The introduction of Rust has not been without contention and controversy from the Kernel Maintainer community (Kroah-Hartman, 2025), highlighting an important need for change management to support key stakeholders for making a polyglot transition, incorporating multiple disparate programming languages into an established codebase.

### Rust vs Memory Safe C/C++

For an established codebase written in C/C++, which heavily relies on established C/C++ libraries, tools, and infrastructure, moving to Rust may not be a quick or cost-effective transition. Memory safety can be achieved in high-performance languages such as C/C++ with the use of Static Analysis tools and fuzzers. Static analysis can improve the reliability and maintainability of the existing codebase by improving code quality and identifying memory safety issues and undefined behaviour. Similar to the Simulator of Theseus approach, static analysis and fuzzers can be implemented incrementally across different parts of the codebase. For instance, putting a focus on specific modules or functions with high risk or importance while leaving other areas of the codebase untouched.

C++'s complexity (e.g., templates, multi-threading, and intricate memory management) can sometimes result in false positives or missed detections during static analysis. C++’s lack of inherent memory safety means that static analysis tools and fuzzers may not catch all potential violations. It may be difficult to get static analysis to work seamlessly across legacy build systems that weren’t originally designed with these tools in mind. While static analysis and fuzzers can help catch memory management issues, it can’t eliminate them entirely.

Static analysis tools for C++, based on compile-time instrumentation, can be expensive and require set-up and configuration to fit the needs of specific projects (Ayushi Sharma, 2024). Some tools have high false positive rates, leading to developer fatigue as they spend time fixing issues that don't address a real problem (Brittany Johnson, 2013). Integrating static analysis into the existing build systems and CI/CD pipelines for large C++ projects can be challenging. It requires expertise to ensure that the tools are configured correctly and that they don’t create too much overhead impacting build times or developer productivity. Some popular static analysis tools for C++ (e.g., Clang Static Analyzer, Coverity, Cppcheck, SonarQube) can have significant overhead in terms of configuration complexity.

### Simulator of Theseus Approach

Rewriting an entire codebase in Rust to realise the benefits of memory safety in a simulator is an enormous task which risks disrupting business continuity. This undertaking does not typically meet the merits for return on investment. The outcome is more secure software, but no immediate customer-facing value. Additionally, short-term productivity gains will be offset by transition costs and learning curves experienced by the business.

To achieve a sustainable Rust transition, an incremental approach is required. This is the core tenet of the Simulator of Theseus approach. Replace the simulation software a component at a time utilising Rust and the system will converge on a memory safe solution, incrementally recognising the benefits of Rust as you transition.
This incremental approach is achieved through Rust’s ability to interoperate with other programming languages through a Foreign Function Interface (FFI). This FFI provides a bridge between the other languages where Rust functions can be called, and Rust can call foreign functions from the legacy code base.

The Simulator of Theseus approach spreads the transition cost over time, where higher risk software areas are addressed first, and new functions or areas of significant change are re-written in Rust. This limits the transition risk, starting small and proving the approach, then increasing the transition aspirations as teams get more comfortable with Rust and managing a polyglot project. This ensures that the business can embark on a transition to memory safe simulators, whilst still retaining the ability to deliver their product and projects.

## HOW TO TRANSITION A SIMULATOR

There are two dimensions to consider when transitioning a simulator, or any complex software system, to Rust. One dimension is technical, answering questions such as what is the design and mechanism of the bridge between Rust and the other software? What are the invariants and constraints imposed on this interface? What is the delegation of responsibility between the rust and legacy software? The other dimension is strategic, answering questions such as what to migrate first and why? How much do you migrate and with what cadence?

Exploring the strategic dimension first, ‘what should be migrated first and why?’, it has been shown in studies that vulnerabilities have a lifetime with an exponential decay (Alexopoulos, Brack, Mühlhäuser, Grube, & Wagner, 2022).

The mean and median of this lifetime is dependent on the language used and the maturity of the security practises within the project, but we can infer a key detail from this result. Vulnerabilities disproportionately exist in new or recently modified code. As such, migrating to Rust should start with any new code or novel development. This ensures that memory safety vulnerabilities are eliminated from this new code, providing a significant improvement to software security. Similarly, any software modifications that invite a significant refactor to implement should also be rewritten in Rust for the same reason.

Prioritising new code being written in Rust will increase the percentage of Rust within the simulator over time, but it is dependent on the cadence of change and addition within the system. To transition more of the simulator, and faster than the default cadence of change, legacy software is required to be reimplemented in Rust. This poses the question of deciding which software should be migrated and with what priority? This question is best answered through a risk-based approach. Which parts of the simulator are likely to have software vulnerabilities that can be eliminated through a migration to Rust?

The probability of code having vulnerabilities that can be eliminated by Rust can be assessed by the factors given in Table 1. Using these factors to assess legacy code, you can identify and prioritise software to transition to Rust. Making determinations of the probability of the existence of vulnerabilities rely on experienced judgement from senior software engineers, but even with imprecision in the assessment these factors can focus and guide a transition priority for legacy software.

**Table 1: Risk Factors for Software Vulnerabilities**

| Factor | Rationale | Indicator(s) |
|---|---|---|
| Age of code | The result from (Alexopoulos, Brack, Mühlhäuser, Grube, & Wagner, 2022) show that vulnerabilities have a lifetime with an exponential decay. It can be inferred that vulnerabilities disproportionately exist in new or recently modified code | Time since the commit for the first release or major refactor of the software component. The smaller this time difference is, the greater likelihood the software contains exploitable vulnerabilities. |
| Complexity of the software | Software that has complex algorithms or non-trivial data structure manipulation make it harder to reason about the software logic and correctness. It can also lead to errors in handling memory or indexing complex data structures due to the cognitive load on the software engineer. These mistakes lead to software vulnerabilities through exploitable memory safety issues. | Software that features complex mathematics involving non-trivial manipulation of data structures or allocation of memory (e.g. physics engines, flight dynamics models). Similarly, software that has poor readability, whether it be through inadequate design or technical debt is not only a candidate for refactoring but also transitioning to Rust. Static Analysis Security Testing (SAST) tools for software code quality and security checks can also be of benefit. Some SAST tools identify security hotspots that require developer review to ensure code correctness. Software with a high number of security hotspots are good candidates for transitioning to Rust. The Rust compiler now performs these checks, rather than a manual check by a developer. |
| Concurrency | Software that manages concurrency can be challenging for a developer to reason about software logic and correctness. Similarly, edge cases in concurrent software can be subtle and not always addressed, leading to data races that are rare and challenging to debug. Data races result in undefined behaviour and can be exploitable and thus make software vulnerable. | Software with threading calls that feature non-trivial concurrent processes (e.g. a simulator real-time scheduler). |

With the transition strategy considered, we move onto the implementation dimension. How will we implement such a transition? Fundamentally, Rust interoperability relies on a Foreign Function Interface (FFI) using a C Application Binary Interface (ABI) to bind to other programming languages (The Rust Programming Language Team, 2025a).

The applicability of being able to transition to Rust depends on the ability for Rust to interoperate with that language. Most modern languages employ an FFI using a C ABI, however older languages may not. Simulators are complex systems that require high performance and low latency. As such, they typically utilise a performant language such as C or C++ for the majority of their software. For the purposes of this paper, we explicitly address the use case of transitioning a C/C++ software component to Rust in a monolithic architecture, which is typical of the Simulators in our organisation.

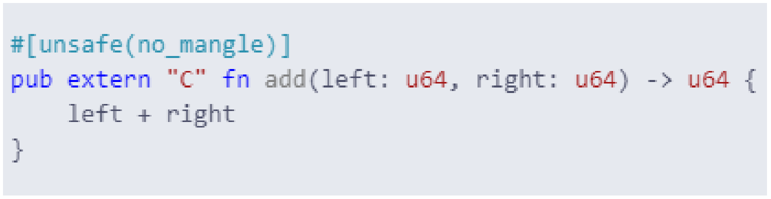
## Foreign Function Interfaces

A foreign function interface (FFI) allows software written in one programming language to call functions or use data written in another language. In the context of integrating Rust with C/C++, a FFI is essential because the two languages may have different calling conventions, data alignment and layouts, name mangling and exception propagation. These language attributes of the compiled binary define the Application Binary Interface (ABI) for that language. FFIs are in common use in industry today, in particular where the performance of low-level languages like C/C++ and Rust need to be called with the convenience of interpreted languages like Python or JavaScript. The Python package NumPy is a good example of a popular software package used in industry making heavy use of FFIs.

NumPy is a fundamental package for scientific computing in Python. NumPy uses pre-compiled C code libraries with an FFI to execute optimised mathematical operations on large datasets. With the core made of pre-compiled C code, much better execution speeds can be achieved without the inefficiencies of interpreting Python code and manipulating Python objects. NumPy allows developers to write software at near-C speeds, with the code simplicity and ease of use expected from Python. (NumPy Developers, 2024)

## Rust Interoperability with C/C++

Fundamentally, Rust provides interoperability with other languages through its FFI. The FFI in Rust generates a binary that adheres to the C Application Binary Interface (ABI) standard. The simplest way to integrate Rust with C/C++ is to use the FFI and write public functions that form the API for the Rust library called from the C/C++ application. These functions require the use of the *extern "C"* keyword and the *no_mangle* attribute to inform the Rust compiler that the function is to use the standard C ABI and to disable the standard symbol name mangling. Because disabling name mangling can lead to symbol name collisions and undefined behaviour, it is required to be invoked using an *unsafe* keyword. The use of the *unsafe* keyword doesn't mean the code is inherently unsafe and shouldn't cause alarm. The unsafe keyword simply means that unsafe operations are permitted in the code block and can potentially violate the memory-safety guarantees of Rust's static semantics. Such sections of *unsafe* code require checking from the programmer that the software's contracts and correctness are sound. In this instance, the programmer is responsible for ensuring that the function name doesn't have name collisions with other symbols in the application. An example of the use of the *extern "C"* keyword and the *no_mangle* attribute with the *unsafe* keyword is shown in Figure 2.

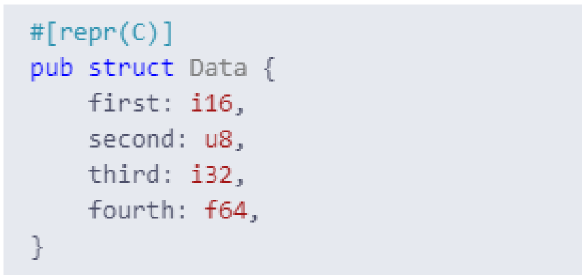


```rust
#[unsafe(no_mangle)]
pub extern "C" fn add(left: u64, right: u64) -> u64 {
    left + right
}
```

**Figure 2: add function using FFI**

Additionally, Rust can be used to create data structures compatible with the C ABI. This is achieved using the *repr* attribute on structs defined in Rust, as shown in Figure 3. This ensures that the memory layout of the data structure is compatible with the C Language and thus can be shared across the FFI between Rust and the C/C++ application.

Using these techniques, you can write a Rust library and expose an API using the C ABI that can be invoked from a C/C++ application. However, with any complex software application it can become quickly tedious to instantiate bindings between Rust components and the rest of the C/C++ application. To solve this problem there are three prominent tools for automatically generating FFI bindings between Rust and C/C++, “bindgen”, “cbindgen” and “cxx”. This saves time and is also less error prone than manually generating these bindings.



```rust
#[repr(C)]
pub struct Data {
    first: i16,
    second: u8,
    third: i32,
    fourth: f64,
}
```

**Figure 3: Data struct using C memory layout**

bindgen is a tool maintained by the Rust Programming Language Team for generating Rust FFI bindings to C and some C++ libraries. It is typically used for generating Rust FFI bindings to existing C and C++ libraries to invoke functions from these libraries within a Rust application or library. It can be used in reverse, providing value in a Simulator of Theseus style Rust transition, first by writing a compatible C/C++ header for the FFI API and then invoking bindgen to generate the Rust bindings to implement the Rust component logic that will be linked into the legacy C/C++ application.

cbindgen is a tool maintained by Mozilla, the company behind the Firefox browser. It produces both C and C++ 11 headers for Rust libraries utilising a public C API. The C++ headers produced by cbindgen allow for a more ergonomic FFI API allowing the use of operator overloads, enum classes and templates. cbindgen can be used as either a standalone program or as a library invoked by the Rust build automation.

CXX is a tool maintained by David Tolnay, a prolific Rust contributor who developed and maintains some iconic and widely used Rust packages (serde, syn, anyhow). CXX is interesting as it takes a novel approach to solving the problem of automatically generating FFI bindings between Rust and C++, and in doing so provides a safe mechanism for invoking the FFI from Rust and C++. CXX achieves this by generating a hidden C API and presenting them to Rust and C++ using a common API semantic as shown in Figure 4. This results in an FFI API that operates with negligible overhead and abstracts the unsafe C-style ABI signatures preventing the pitfalls associated with using a C ABI FFI.

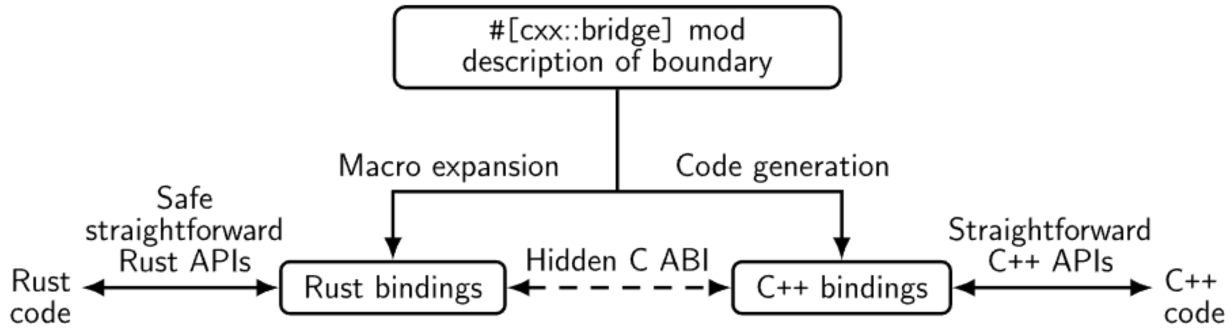


**Figure 4: CXX FFI Bridge (Tolnay, 2025)**

## Challenges and Considerations

When executing a transition to Rust it is important to start small and consider components that are low-hanging fruit first. This ensures that the risk and consequence are minimal while the business comes up the learning curve of integrating Rust and managing a polyglot code base. A Rust transition can take months or years depending on the complexity of the software and cadence of change, so achieving quick wins early and getting momentum for the transition is important.

An additional consideration is Change Management. It is important to bring software engineers on the journey to Rust and managing a polyglot code base. Such a change in software development can be quite divisive if not managed transparently with adequate support. Such challenges are evident in the Rust for Linux initiative, causing some friction between the Rust advocates and the legacy C maintainers (Kroah-Hartman, 2025). Identifying key advocates within the organisation and establishing a support network for Rust is important. Formal training in Rust will be a benefit but critically having vocal advocates for change and empowering them to support the software engineering community with the transition, through coaching and communication, will be instrumental.

As with any significant refactoring, adequate unit test coverage of the software component interface with the rest of the solution is critical (Fowler & Beck, 2019). If the component to transition lacks unit test coverage, it is important to instantiate it. This ensures that when a software component is replaced with a Rust implementation, the unit tests detect any inevitable mistakes, and that the software still functions as per the contract with the rest of the solution. Depending on where the boundary of the FFI is between the logical components within the solution, these checks may be performed on the Rust side, or on the C/C++ side or a mixture of the two.

On the technical side, it is important when specifying an FFI to minimise the boundary. Fewer and simpler functions across the FFI is safer and less prone to errors. Additional considerations should be made around the ownership of memory across the FFI boundary. What are the lifetimes on the memory and invariant rules on how memory is to be managed across the FFI boundary? Such considerations are important to establish early in the FFI design. Here the use of CXX could be an attractive choice as it provides an intentionally restrictive and opinionated FFI bridge to deliver an expressive set of functionalities providing safety guarantees for the FFI.

Establishing an FFI will involve the use of the *unsafe* keyword. As explained earlier, this doesn’t mean the code is inherently unsafe, instead checking the unsafe code sections for soundness becomes the programmer’s responsibility. Here the definition of soundness is the property of never causing undefined behaviour when invoked from arbitrary safe code, even in combination with other sound code (The Rust Programming Language Team, 2025d). The Rust language provides a tool called Miri for automated checking of undefined behaviour. It employs the same techniques typical of SAST, analysers and fuzzers instrumenting code for runtime checks for soundness. It leverages Rust’s mid-level intermediate representation and provides checking for memory and type violations that leads to undefined behaviour. Like all SAST, sanitisers and fuzzing tools, Miri cannot ensure code is sound but builds confidence of soundness given quality test coverage. Rust’s strength is that most code written will be in safe code, which has safety guarantees provided by the Rust Compiler. Limiting the use of unsafe code blocks to only where needed significantly reduces the extent to which code must be scrutinised for soundness.

Another technical consideration is managing Rust panics. Rust libraries integrated with a C/C++ application using an FFI should not issue a panic. A panic in Rust is a runtime signal indicating that the program has encountered an unrecoverable error and signals that the application should unwind and exit. If the Rust library utilised by the FFI can issue a panic, it is required to be caught by the same rust library and lead to termination of the application. Conveniently, Rust functions defined using extern “C” ensures that if Rust code panics it is automatically caught, and the process is aborted (The Rust Programming Language Team, 2025b). Letting the Rust function unwind into the C/C++ runtime leads to undefined behaviour. More information about the ABI and function unwinding can be found in the Rust Reference documentation (The Rust Programming Language Team, 2025c).

Similarly, consideration needs to also be given to how to manage error handling across the FFI boundary. C, C++ and Rust all have different error handling conventions which need to be intercepted at the FFI boundary and translated to the receiving language’s convention when propagated across the boundary. Conveniently, CXX provides error conversion automatically as part of its FFI Bridge implementation which may be desirable as it removes this concern from the FFI design.

These are the key challenges requiring consideration in a Rust transition plan and the design of an FFI between Rust and the rest of the C/C++ application. Getting these details right early avoids pain later in the implementation and improves the probability of success in realising an incremental Rust transition to achieve memory safety in simulation software.

## PROOF-OF-CONCEPT

A classic rigid double pendulum was modelled using a proprietary real-time framework in C++ and performance benchmarked before refactoring with the Simulator of Theseus approach, converting the individual C++ simulation modules to use Rust via an FFI. Cbindgen was used to enumerate the FFI binding between the Rust library and the C++ real-time framework. This generator was chosen due to its simplicity to implement on our simulation modules and their interface to the real-time framework. CXX was considered but due to its opinionated FFI Bridge approach abstracting the C ABI, it was a heavier modification to the C++ real-time framework compared with a bindgen or cbindgen implementation. Future work would analyse the three FFI generators for interoperability between C/C++ and Rust, assessing their performance across key factors such as computational cost, simplicity to implement, readability and maintainability.

The double pendulum was chosen as a classical physics demonstration of chaos in a system (Shinbrot, 1992), small changes to initial conditions have wildly different outcomes, where the equations of motion have known derivations using the Euler-Lagrange equation (Herho, 2024), or Newtonian physics. Rewriting existing proprietary simulator components in Rust were avoided in this study to isolate performance timings to a Rust versus C++ implementation. This removes any issues arising from converting complicated simulation business logic, obscuring the performance comparison of C++ and Rust with an FFI.

The simulation software consisted of two components, one for each link of the double pendulum. The simulation component logic calculated the acceleration of the link and then calculated velocity and displacement by multiplying the acceleration by time delta. The simulation was run for a duration of 140 seconds with a frame timing of 250hz. The simulation was repeated 20 times to get an average of frame timings and baseline expected variance. The Simulator of Theseus approach was used to introduce a Rust library to calculate the pendulum physics. The C++ real-time framework was still used to orchestrate the simulation, but the real-time modules now using functions from the Rust library via the FFI.

**Results**

The runtime for each component during each frame of the simulation was recorded and averaged over 1 second intervals. Figure 5 plots this data for both the C++ implementation and Rust implementation. Superimposed on this data, two lines show the mean runtime the simulation component took across all simulation iterations.

The graph shows some variability over the total simulation time for the mean frame runtime for both the C++ and Rust implementations. The min, max, mean, standard deviation and variance of the mean frame runtime are listed below in Table 2. The variance between C++ and Rust are comparable. The Rust plot shows a slightly longer mean frame runtime by approximately 20μs.

The slight increase in mean frame runtime for the Rust implementation is likely attributed to the FFI function call to the external Rust library. This increase in mean frame runtime is negligible and is acceptable for real-time simulation software. However, it does highlight the need for planning in the design of the FFI to ensure foreign calls aren't haphazardly and repeatedly invoked leading to an accumulation in computational delays. Typically, these foreign functions cannot be inlined to avoid the cost of a context switch, so their use needs to be planned in the context of the software architecture for latency sensitive software.

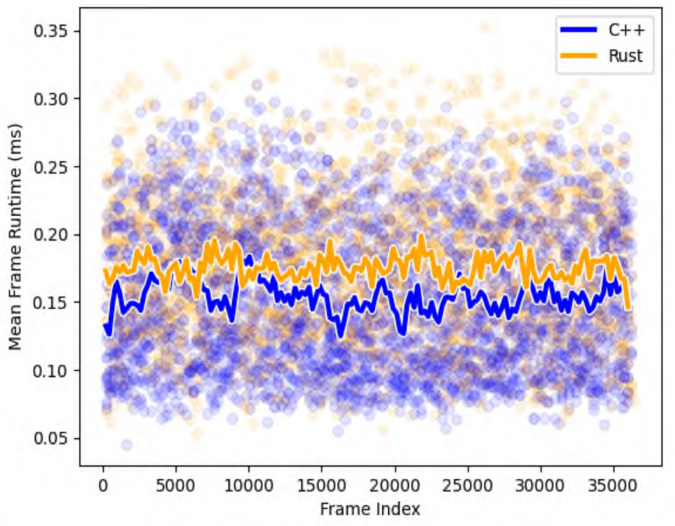


**Figure 5: Proof-of-Concept Mean Frame Runtime across the 20 Iterations**

**Table 2: Mean Frame Runtime Performance across the 20 Iterations**

| | **C++ (ms)** | **Rust (ms)** |
|---|---|---|
| Min | 0.045 | 0.054 |
| Max | 0.312 | 0.352 |
| Mean | 0.155 | 0.175 |
| Std Dev | 0.054 | 0.053 |
| Variance | 0.003 | 0.003 |

## CONCLUSION AND FUTURE WORK

Due to the growing sophistication in cyber threats and the importance of simulators providing training outcomes to maintain critical industries, simulators require cyber resilience. Software vulnerabilities are a vector of cyber-attack for simulators, of which memory safety vulnerabilities are disproportionately responsible for software vulnerabilities in software systems. It is paramount to eliminate or minimise memory safety vulnerabilities in simulation systems to greatly reduce this attack vector.

Through a transition to using the Rust programming language in simulation systems, memory safety defects can be eliminated through the safety guarantees provided by this language at compilation, enabling significant improvements in the cyber resilience. Such a transition to Rust can be achieved through an incremental replacement strategy whereby high-risk areas are migrated first and over time the system converges on a memory safe implementation. This is the core tenet of the Simulator of Theseus approach which amortises the transition cost overtime, managing the risk of transition and allowing the system to recognise the benefits provided by Rust incrementally.

The benefits using Rust are substantial and go beyond memory safety, delivering improvements in programming productivity and the elimination and early identification of many classes of software defects. Rust is a performant language suitable for the demands of real-time simulation systems, and the impacts on performance integrating Rust into a traditional C/C++ simulation system are negligible. This was shown through the proof-of-concept that explored the integration of Rust into a proprietary real-time framework used for simulation systems.

Beyond this paper, future work would analyse the three FFI generators for Rust and C/C++ interoperability and assess their performance across key factors such as their computation cost to implement, simplicity to implement and their readability and maintainability. Recommendations on which generator to use in which scenario would be determined as part of this analysis. In addition, the transition of a simulation component from a production environment to Rust and its integration in a configuration typical of deployment would be explored. Performance and benchmarking would also be assessed. Lastly, a deeper investigation into the CXX generator would be conducted to explore the design and integration of its FFI Bridge into a C++ real-time simulation framework, and how the C ABI abstraction contributes to memory safety within simulation software undergoing a Rust transition.

## REFERENCES

Alexopoulos, N., Brack, M., Mühlhäuser, M., Grube, T., & Wagner, J. P. (2022). How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes. *USENIX Security Symposium.* Boston, MA, USA: USENIX.

Ayushi Sharma, S. S.-A. (2024). Rust for Embedded Systems: Current State and Open Problems. *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, (pp. 2296-2310). Salt Lake City, UT, USA. ACM, New York, NY, USA.

Brittany Johnson, Y. S.-H. (2013). Why Don't Software Developers Use Static. *2013 35th International Conference on Software Engineering (ICSE)*, 672-681.

Burton, J., Janjeva, A., Moseley, S., & Alice. (2025). *AI and Serious Online Crime.* London, UK: Centre for Emerging Technology and Security, The Alan Turing Institute.

Cybersecurity and Infrastructure Security Agency. (2023). *The Case for Memory Safe Roadmaps.* Virginia, United States: US Department of Defense.

Fowler, M., & Beck, K. (2019). *Refactoring: Improving the Design of Existing Code* (2nd ed.). Indianapolis, Indiana, US: Pearson Addison-Wesley.

Herho, S. F. (2024). Reappraising double pendulum dynamics across multiple computational platforms. *HAL Open Science*, hal-04568479.

Klabnik, S., & Nichols, C. (2023). *The Rust Programming Language.* San Francisco, California: No Starch Press.

Kroah-Hartman, G. (2025). *Re: Rust kernel policy*. Retrieved from The Linux Kernel Archives: https://lore.kernel.org/rust-for-linux/2025021954-flaccid-pucker-f7d9@gregkh/

Microsoft. (2025, May 15). *Rust for Windows*. Retrieved from Github: https://github.com/microsoft/windows-rs

MITRE. (2025). *CWE CATEGORY: Comprehensive Categorization: Memory Safety*. Retrieved from Common Weakness Enumeration: https://cwe.mitre.org/data/definitions/1399.html

MSRC Team. (2019). *A proactive approach to more secure code*. Retrieved from Microsoft Security Research & Defense: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

NumPy Developers. (2024, December 14). *What is NumPy?* Retrieved from NumPy: https://numpy.org/doc/stable/user/whatisnumpy.html

Ojeda , M. (2021). *[RFC] Rust support*. Retrieved from https://lwn.net/Articles/852659/

Resecurity. (2024). *The Aviation and Aerospace Sectors Face Skyrocketing Cyber Threats*. Retrieved from Resecurity: https://www.resecurity.com/blog/article/the-aviation-and-aerospace-sectors-face-skyrocketing-cyber-threats

Shinbrot, T. G. (1992). Chaos in a double pendulum. *American Journal of Physics, 60(6),* 491-499.

The Rust Programming Language Team. (2025a). *Foreign Function Interface*. Retrieved from The Rustonomicon: https://doc.rust-lang.org/nomicon/ffi.html

The Rust Programming Language Team. (2025b). *Function catch_unwind*. Retrieved from The Rust Standard Library: https://doc.rust-lang.org/std/panic/fn.catch_unwind.html

The Rust Programming Language Team. (2025c). *Functions*. Retrieved from The Rust Reference: https://doc.rust-lang.org/reference/items/functions.html

The Rust Programming Language Team. (2025d). *Miri: an interpreter for Rust's mid-level intermediate representation*. Retrieved from Github: https://github.com/rust-lang/miri

Tolnay, D. (2025). *Introduction*. Retrieved from CXX — safe interop between Rust and C++: https://cxx.rs/index.html

Torvalds, L. (2022). *Linux 6.1*. Retrieved from The Linux Kernel Archives: https://www.kernel.org/pub/linux/kernel/v6.x/ChangeLog-6.1

Weston, D. (2023, April 19). *Windows 11 Security by-default.* Retrieved from GitHub: https://github.com/dwizzzle/Presentations/blob/master/David%20Weston%20-%20Windows%2011%20Security%20by-default%20-%20Bluehat%20IL%202023.pdf