Foundations for a Rust-Like Borrow Checker for C

Tiago Silva
University of Porto
Porto, Portugal
tiagodusilva@gmail.com

João Bispo
University of Porto / INESC TEC
Porto, Portugal
jbispo@fe.up.pt

Tiago Carvalho
Polytechnic Institute of Porto, School of Engineering / INESC TEC
Porto, Portugal
tdc@isep.ipp.pt

Abstract

Memory safety issues in C are the origin of various vulnerabilities that can compromise a program's correctness or safety from attacks. We propose a different approach to tackle memory safety, the replication of Rust's Mid-level Intermediate Representation (MIR) Borrow Checker, through the usage of static analysis and successive source-to-source code transformations, to be composed upstream of the compiler, thus ensuring maximal compatibility with most build systems. This allows us to approximate a subset of C to Rust's core concepts, applying the memory safety guarantees of the rustc compiler to C. In this work, we present a survey of Rust's efforts towards ensuring memory safety, and describe the theoretical basis for a C borrow checker, alongside a proof-of-concept that was developed to demonstrate its potential. This prototype correctly identified violations of the ownership and aliasing rules, and accurately reported each error with a level of detail comparable to that of the rustc compiler.

CCS Concepts: • **Software and its engineering → Software safety**; **Preprocessors**; **Allocation / deallocation strategies**.

Keywords: C, Rust, Source-to-Source, Memory Safety, Static analysis, Borrow checker, Lifetimes, Ownership, Transpiler, Code transformations

ACM Reference Format:
Tiago Silva, João Bispo, and Tiago Carvalho. 2024. Foundations for a Rust-Like Borrow Checker for C. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '24), June 24, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3652032.3657579

LCTES '24, June 24, 2024, Copenhagen, Denmark

ACM ISBN 979-8-4007-0616-5/24/06
https://doi.org/10.1145/3652032.3657579

1 Introduction

Performance and flexibility regarding the representation and manipulation of memory are some of the most prized aspects of the C programming language. The latter, however, constitutes a dangerous double-edged sword, as they can be a source of various bugs [26] that result in memory safety vulnerabilities that are almost exclusively found in C/C++ [31]. Previous research into this topic has proposed techniques such as the application of static analysers, the insertion of runtime memory checks, or the introduction of special annotations leading to new dialects such as Cyclone [33].

Historically, multiple approaches have been proposed to detect and correct these safety issues over the years. One of these solutions was the creation of a type-safe language whose compiler guaranteed memory and concurrency safety, which ultimately became the basis for the Rust programming language [14].

Although new projects could adopt Rust to become more secure, the C programming language remains the primary choice for safety-critical projects restricted by formally verified compilers, programs targeting embedded systems with poor Rust compilation toolchains, or even legacy codebases. The process of automatically converting from C codebases to Rust remains an arduous process which is the target of various proposals, such as Ling et al. [17].

In this work, we present an approach to replicate Rust's borrow checker in C, allowing for memory safety to be enforced at compile-time, without the need for heavy runtime overhead. Our source-to-source approach, based on the Clava [3] compiler, resulted in a compiler agnostic solution, allowing it to be easily integrated into existing build systems.

This proposal laid the required groundwork for a fully-fledged C borrow checker, based on a survey of Rust's design, functionalities and internals. We highlight the most relevant algorithms that safeguard against dangerous pointer aliasing, and present an early proof-of-concept prototype to demonstrate the viability of a C borrow checker.

During the background section, we present a small introspection into Rust's core principles and foundations, from single ownership and aliasing, to a brief introduction of the responsibilities of its borrow checker. In the following section, we catalogue the work performed on memory safety, and identify how our work distinguishes itself from previous research. Afterwards, we present the theoretical foundations for a C borrow checker, with a focus on automatic memory

management through the use of annotations and `#pragma` directives, alongside its possible limitations. In the implementation section, we describe the internals of our proof-of-concept, as well as present the results obtained through its testing on various synthetic programs. Finally, we present our conclusions and delve into the future work required to both polish and expand the proposed C borrow checker.

2 Background

2.1 Ownership and Borrowing in Rust

Although Rust has no official pointer aliasing model, there have been proposals to formalize it by Jung [13] [12]. Rust relies on a strict usage of pointer aliasing, in particular regarding to mutable references, expressed through the concepts of ownership and borrowing. According to the authors of Rust [14], the three core rules of data ownership are:

- Each value in Rust has an *owner*
- There can only be one owner at a time
- When the owner goes out of scope, its value is dropped

The goal behind these rules is to ensure that data is not invalidated due to freeing a resource too soon, and that no Undefined Behaviour (UB) occurs due to freeing the same memory more than once.

Rust addresses these with its ownership model, as memory should be automatically returned once the variable that owns it goes out of scope. To achieve this, Rust uses the `Drop` trait, to indicate that a given data type has resources that must be cleaned up upon exiting its scope. This trait is used to clear heap memory or close network connections, among others. Foremost, this approach automatically avoids memory leaks, by ensuring that any memory or resource is always freed. In addition, thanks to the memory only being freed *after* its owner goes out of scope, and thus, no longer accessible, data will never be invalid. As for freeing the same resource twice, Rust's ownership model ensures that each resource has a single owner, therefore making it impossible to free the same resource more than once. These mechanisms allow Rust to detect violations of temporal safety at compile time.

When operating with ownership, an assignment such as `let foo = bar` may have two meanings, be it either as a copy or a move operation. The compiler uses its strong typing system to distinguish between them. It copies basic data types such as integers, or structs that implement the `Copy` trait, that is, data that can be safely copied with no side-effects. Likewise, immutable references can also be safely copied, as they will never be able to mutate the original data. These copies guarantee that the original value remains valid. On the other hand, a move operation takes the ownership from one variable, and assigns it to the other. After this change of ownership, the original variable is no longer valid, and its variable becomes inaccessible. This ensures that one and only one owner of the data exists at any given time.

This can be extended to move values across functions, where the ownership of a value can be transferred into the function, in which it may be modified, and then possibly returned back to the caller. As most large data types are stored in the heap, this operation is extremely efficient, as it only requires the transfer of the stack-allocated data, such as a pointer and some bounds information.

In order to simplify the syntax and improve the semantics of Rust, the mechanism for borrowing was introduced to transfer temporary ownership of variables, a mechanism comparable to C++ references. Unlike traditional pointers, however, references must ensure to "point to a valid value of a particular type for the life of that reference" [14], that is, they must always point to a valid object or function.

2.2 The Rust Borrow Checker

The borrow checker is the core system that ensures compile-time safety guarantees in Rust, and we can consider that it has seen, up until now, three main iterations: Abstract-Syntax Tree (AST) Borrow Checker, Non-Lexical Lifetimes (NLL) [21] (later renamed to MIR Borrow Checker), and Polonius [30]. Although the third and newer system was announced in 2019 [22], in this work we have opted to instead focus on the previous system, the Non-Lexical Lifetimes. This system is fully stable since 2022 [23], which means that it has been extensively tested by the community, and is much better documented.

The borrow checker introduces the concept of lifetimes to guarantee the single ownership property. In other words, to ensure that values are not mutated or moved while they are borrowed elsewhere. A lifetime is created anytime a loan occurs, and it corresponds to the (possibly non-continuous) span of code - or region - in which a reference may be used. However, this term can be ambiguous, as it can also refer to the lifetime of a variable - or scope - which is the span of code before that variable is freed. Nonetheless, both definitions are intertwined, as the scope of a variable must always outlive all of its references. Contrarily, references pointing to freed memory could lead to use-after-free bugs.

To summarize the NLL algorithm as described in its respective Request For Comments (RFC) [21], it can be subdivided into the following steps:

1. Create a region variable (containing a set of points, or code statements) to represent each lifetime involved
2. Build constraints from liveness analysis, subtyping and variance rules, and reborrows
3. Propagate the lifetimes through an inference algorithm
4. Detect and report any violations of the borrowing rules

This system was only possible thanks to the prior introduction of the Mid-level Intermediate Representation in 2015 [19], a new IR that rests between Rust's AST and the generated LLVM bytecode. In short, it represents each function individually as its own Control-Flow Graph (CFG). This

granted two advantages to the NLL proposal, first to only support a much more restricted and normalized instruction set than surface-level Rust, and second, to better synergize with the nature of the algorithms required for liveness analysis, constraint propagation, and more robust drops. In regards to the latter, a separate algorithm named drop elaboration [15] is responsible for the introduction and management of any calls required to free allocated resources.

3 Related Work

The research addressing the memory safety of C is very extensive, with papers from the 1980s still being quite relevant today. In Checked C [6], these techniques are categorized in four groups: "C(-like) dialects, compiler implementations, static analysis, and security mitigations".

3.1 Dialects and Extensions

One of the solutions to the lack of memory safety in C, is to use a different language, or to by introducing changes to the language itself. DeLine et al. introduce Vault [4], a programming language with statement and expression syntax based on C, with an additional system that allows for types to be accompanied by a type guard, which consists of zero or more atomic predicates, each of which is a simple check for the existence of any given key in an abstracted global state that Vault keeps track of. These ideas are not dissimilar to Rust's ownership model, both of which are performed exclusively at compile time.

Cyclone [33] is an extension of C, it uses the same preprocessor, and largely follows most of C's lexical conventions and grammar. The major changes include the addition of static analysis, the insertion of runtime checks for operations whose safety cannot be determined during the compilation, and custom annotations to supply hints to the static analysis. An example of these annotations is the "never-NULL" pointers, indicated with @ instead of $*$. In general, Cyclone aims to make program security mandatory, instead of offering safe, but optional, checks which were often ignored by programmers. It accounts for buffer overflows, uninitialized pointers, and more. More relevant to this work is their *region analysis* [9], which allows Cyclone to perform an intraprocedural region analysis to accurately reject programs that return dangling pointers, in a way that is not too dissimilar to what is done in Rust.

CheckedC [6] extends C with two *checked pointer types*, `_Ptr<T>` and `_Array_ptr<T>`, in order to protect against data modification and disclosure attacks. The first indicates that a pointer can only be dereferenced, and no arithmetic will be performed on it, while the second supports pointer arithmetic with additional bounds declarations. These checks are deferred to runtime, exchanging runtime overhead for coding flexibility. Within special blocks or functions marked as *checked regions*, the compiler also imposes stronger restrictions on the usage of unchecked points that could corrupt the checked ones through aliasing.

3.2 Compiler Implementations

Instead of exploring the avenues of new languages or language extensions, safe compiler implementations focus on methods to improve the spatial security of existing C code. Many works in this category change the representation of pointers, effectively creating fat pointers, or "marking" regions of allocated memory.

SoftBound [25] consists of compile time transformations for enforcing spatial safety in C, inspired by a previous hardware-assisted approach, HardBound [5]. SoftBound introduces metadata to track the base and bounds of pointers, without requiring any changes to C source code, but achieves this strictly at a software level, and performs the metadata manipulation whenever storing or loading pointer values. Furthermore, a formal proof showed that SoftBound is able to ensure spatial safety for any program, even in the presence of arbitrary casts. These guarantees come at an average cost of a 67% higher runtime overhead, but offers a store-only checking mode that fully passed their test suite while diminishing the overhead to 22%.

Write Integrity Testing (WIT) [1] is a technique to provide protection against attacks attempting to exploit memory errors. It uses a points-to analysis at compile time to compute the CFG and the set of objects that can be written by each instruction in the program. WIT then generates code to prevent instructions from modifying any object not present in the set of writable objects, computed through a static analysis, and also to ensure that any indirect control transfers are allowed by the CFG. These techniques could be applied to C or C++ programs without requiring any source code modifications, with an average runtime overhead of 7%.

In Purify [10], Hastings et al. propose the use of a bit table that holds a two-bit state code for each byte in the heap, stack, data, and bss. The possible states are 'unallocated' (unwritable and unreadable), 'allocated but uninitialized' (writable but unreadable), and 'allocated and initialized' (writable and readable). In other words, a byte-level tagged architecture, where tags represent the memory state. The bit table maintained by purify results in a 25% memory overhead during development.

3.3 Static Analysis

Static program analysis aims to avoid the run time overhead of the various safe C compiler implementations mentioned above. These tools analyse source-code or binaries to find vulnerabilities and bugs, such as out-of-bound-arrays accesses or incorrect pointer arithmetic.

Nevertheless, such systems face a different set of challenges, such as scalability and false positive management. As

codebases grow in size, static analysis needs to balance performance and precision. A precise analyser may not be able to scan large programs [6]. In commercial environments this is especially critical, as many companies run static analysers as part of a nightly pipeline, and if the analyser is unable to produce meaningful results within that time frame, its effectiveness is questioned.

Coverity [2] explored the management of false positives, which can be more problematic than it initially appears, as people tend to disregard issues found as false positives, thus allowing true bugs to slip by. As such, a cycle of low trust starts building, causing complex bugs to be labelled as false positives, which feeds back into this cycle. Coverity faced this problem first hand, through their years of experience making their static analyser commercially viable. In their words, "when forced to choose between more bugs or fewer false positives, we typically choose the latter". Their aim is to always maintain a false positive rate below 20% in their stable checkers.

An approach that attempts to emulate Rust will fit into this category, as it will perform static analysis over the source code, however it addresses these two problems. On one hand, by having well-defined rules regarding the lifetimes of function parameters and return values, the analysis can be exclusively done at an intra-procedural level, making it scalable. On the other hand, a Rust-like approach is much more restrictive regarding the kind of code it accepts, even code that otherwise could be considered safe, which in practice means trading false positives with false negatives.

3.4 Security Mitigation

Security mitigations are exclusively runtime mechanisms able to either detect whenever memory is corrupted, or prevent attackers from manipulating systems after such corruptions. Some of these techniques are already prevalent in widely adopted compilers such as gcc or Clang.

On the one hand, mechanisms such as address-space layout randomization, control-flow integrity, and data execution prevention, focus on the detection and prevention of arbitrary code execution and control-flow modification. On the other hand, stack protection mechanisms, like stack canaries or shadow stacks, focus on the protection of data and return addresses on the stack.

4 Foundations for a C Borrow Checker

We propose a system to replicate in C most of the safety guarantees provided by Rust's borrow checker. This can be achieved through a source-to-source approach, based on static analysis of annotated source code, and the generation of a new version of the source code with the changes necessary to uphold the desired safety guarantees. Consequently, a method to replicate the ownership semantics of Rust is also required, but we propose that this can be achieved exclusively within the syntax already available in C.

We focus exclusively on static analysis, so we do not describe the addition of runtime safety checks to the generated code, such as bounds or overflow checks, as they could theoretically be achieved by layering previous work in this domain on top of the code output by the C borrow checker.

We have decided to target the C99 standard [11] for the output code, in order to take advantage of the `restrict` qualifier, first introduced by it, as well as to ensure a high degree of retro-compatibility with other compilers and build toolchains.

4.1 Normalization

Although operating over a source-to-source compiler already provides an intermediate representation to work with, fundamentally, reasoning with the entirety of C syntax it still required. As such, we normalize the code into a simpler form, in order to simplify the analysis, through a series of transformations applied to the code before the main analysis begins. Matos [18] has concluded that several normalization steps can be safely performed without affecting the final performance of the compiled code, removing the need to reverse any of the transformations. We aim to normalize the code into a form similar to that of Rust's MIR, which includes the following transformations:

- Break up variable declarations, such that each statement only declares a single variable
- Remove variable shadowing (i.e. redeclaring a variable with the same name in an inner scope)
- Remove operations with side effects such as `++` and `--`, and replace them with their equivalent `+= 1` and `-= 1` operations
- Replace assignments such as `+=` and `-=` by their explicit form, such as `a = a + b`
- Ensure each *rvalue* of every assignment is either:
  - A variable or immediate
  - At most one unary operator applied in succession to a variable or immediate
  - A single binary operator between two variables or immediates
  - A single inter-procedural call, whose arguments are either variables or immediates
- Normalize `x->y` arrow operator into parenthesis and field access operator `(*x).y`

4.2 Memory Management

One of the key aspects of a proper borrow checker is ensuring that all resources are appropriately freed. In Rust, the `Drop` trait automatically guarantees that any allocated resources are freed when a variable goes out of scope or is reassigned. We took inspiration from the drop elaboration process of Rust, and applied a version of it to C, effectively

simulating the RAII (Resource Aquisition Is Initialization) pattern, popular in languages such as C++. With this in mind, Rust's drop systems are built on top of a set of assumptions that we must also safeguard in C, if we are to implement a similar solution, some of which are, in no particular order:

1. Structs cannot be partially initialized
2. Structs that implement `Drop` cannot have data moved out of their fields
3. References in fields must always point to valid memory at the moment of the drop, except for fields marked with `[may_dangle]`
4. Null references are not allowed

Any violations of the first two points can be detected by a pass iterating over every assignment. However, as restricting C to force the complete initialization of long structs would be undesired, we can provide builder functions or define syntactic sugar for our borrow checker. Although neither language has the concept of "objects", Rust has also opted for a similar solution, for example, with the `Default` trait or through the ability to initialize uninitialized fields from another `struct`. In spite of C not defining any syntax for such operations, we can take advantage of the level of access that source-to-source provides to introduce `#pragma` directives that will initialize any missing fields to the desired values.

Let us assume that a `struct A` contains a pointer to another `struct B`. The third assumption ensures that any attempt to free the resources allocated by `B` is always safe, as if the inner pointer was pointing into invalid memory, it could result in reading into an invalid address. Arguably, the most difficult assumption to maintain for our C borrow checker would be the complete removal of null references, due to the predominant usage of NULL in C, especially throughout its standard library, and merits future consideration. Our current plans to address this issue lie in TypeScript-like type annotations outside of the C type system (e.g., `Object | undefined`), in order to distinguish between a maybe-null "raw" pointer, and a safe memory reference that is ensured by the borrow checker. This would also serve as a proxy to Rust's boundary between "safe" and "unsafe" code.

4.3 Heap Memory

Within safe Rust, most interactions with heap memory are performed through abstractions like `Box<>` or other structs that use boxes internally, which ensures that the allocated memory is always freed at the end of a code block. As a simplification, we may consider a `Box<>` as a simple wrapper of a pointer to a heap-allocated value. When creating the box, that memory is allocated, and when it goes out of scope or is reassigned, the implementation of `Drop` ensures that its memory is returned to the allocator. Moreover, the `Box` is an incredibly useful and prevalent concept thanks to its generic nature, which allows it to easily encapsulate any type.

The first major hurdle of attempting to transition such a concept into C is the lack of generics. In order to circumvent this issue, we propose the creation of a wrapper `struct` for each type we wish to box, and further utilize macros to automatically generate the necessary wrapper code for each type. In Rust, a `Box` always has the `Drop` trait associated with it, so the language automatically guarantees the safe cleanup of that allocated memory. For the C borrow checker, we will need to generate and introduce the proper calls to `free`.

Furthermore, Rust disallows any direct calls to the allocator outside of unsafe code, and instead encourages the use of the provided smart pointers, such as `Box`, `Rc` and `Arc`, or the `Vec` data structure. While we can provide generators for specific instances of the smart pointers, `Vec` is very different, due to requiring its allocated memory to be resized dynamically.

4.4 Replicating Rust Syntax through Annotations

As C does not have concepts such as ownership, borrowing, or lifetimes, we need to introduce new syntax in order to provide the information required for our analysis. Conversely, we do not wish to create a dialect of C, as it would go against our goal of retro-compatibility with existing compilers, and of preserving idiomatic C code. As such, we make extensive use of annotations through `#pragma` directives to provide additional information to the compiler, a common practice in C compilers.

Firstly, to replicate the ownership and borrowing syntax of Rust, we leveraged the built-in type system and the `restrict` qualifier. In short, it indicates that any data directly or indirectly accessible through a given pointer `p` can only be accessed through `p`, or a pointer derived from it. Paraphrasing from the C99 standard [11], let P be a restrict-qualified pointer to a type T, and B denote the block it was declared on. Particularly important is the *based on* definition, which states that a pointer expression E is *based on* P if (at some sequence point in the execution of B prior to the evaluation of E) modifying P to point to a copy of the array object into which it formerly pointed would change the value of E.

Whilst this keyword may be ignored by some compiler implementations, it can be leveraged to generate more optimized assembly. In most C code, it is rare to find uses of `restrict`, mainly due to how predominantly loose the use of pointer aliasing is. In turn, this drastically limits its applicability, as if pointer aliasing does occur, it constitutes UB. Even so, it fits perfectly within the scope of this work, as we can ensure that, by definition, there can only be one mutable borrow in-use at any given point of a program's execution.

Interestingly, in the paper regarding the Stacked Borrows aliasing model, Jung [13] confirms that "the Rust compiler (which uses LLVM as its backend) used to emit the LLVM equivalent of restrict as annotations for mutable references in function argument position". This establishes a link between the aliasing rules of Rust and LLVM's `noalias`, which

in turn, according to LLVM's documentation on parameter attributes, is intentionally based on C99's `restrict` qualifier for function parameters, with some differences regarding the effects of returning a `restrict`-qualified pointer.

Given the link between the `restrict` qualifier and mutable borrows, it is only logic for us to reflect this on our syntax, and represent every mutable borrow as a `* restrict`-qualified pointer. Or at least, so we would wish, as the dangers of not respecting the `restrict` qualifier would lead straight to UB. The C99 standard [11] was unclear in its specificity of which behaviours were standardized, blurring the lines of what constitutes UB, and it would appear that this has not changed ever since it was first introduced.

Once again, Jung [13] also arrived at the conclusion that the semantics of `restrict` are unclear, in particular outside of function parameters. Historically, this matter has been error-prone, and led to reported bugs in LLVM [8, 27], even in cases that closely followed the formal definition. Although these bugs have since been patched as of LLVM 12, they had repercussions on the Rust ecosystem, which had to temporarily halt the use of the LLVM `noalias` semantics, as the bugs led to incorrect results, which were even reproducible in C code[1]. Ultimately, it is nearly impossible to predict the behaviour of `restrict` for use in general pointers. To make the situation even more dire, there is a clear lack of tools to check for violations of the `restrict` properties.

To summarise, the utilization of the `restrict` qualifier would be ideal, as it could allow compilers to theoretically optimize programs accepted by our borrow checker, and it would also follow the guidelines of Rust, by requiring explicit declarations of variable mutability. However, due to the lack of a clear definition of its semantics, as well as the lack of tools to check for possible violations, they should be handled with extreme caution. Fortunately, as our solution is implemented at a source-to-source level, so we can freely manipulate the pointer qualifiers to our advantage. As such, we suggest three avenues to explore in the future, with different degrees of conservativeness, with the definition of mutable borrows through `* restrict` pointers in common:

- Keep every `* restrict` pointer as-is, and hope that the compiler can highly optimize the code, but at the significant risk of generating UB
- Keep `* restrict` pointers as-is for function arguments, and remove the qualifier for all other cases, which should still lead to some optimizations, but drastically reduce the chance of UB
- Remove every `* restrict` qualifier, removing a useful vector for program optimization, constituting the least performant, but safest, option

In regards to the annotation of struct traits, whilst our aim is not to replicate the entirety of the trait system, at the very minimum, a representation is required for the traits that inform the borrow checker about the behaviour of a `struct`. In cause are the `Drop` trait for safe memory management, and the `Copy` trait, for flexibility and performance.

[1] https://github.com/rust-lang/rust/issues/54878

Listing 1. Simple function demonstrating the syntax for explicit lifetime annotations

```
#pragma lifetime x %a
#pragma lifetime y %b
#pragma return_lifetime %a
const int *bar(const int *x,
               const int *y) {
  return x;
}
```

Lastly, annotations are also required for named lifetimes throughout the code, mainly required by function parameters, and return values with reference types or `struct` declarations. An example of these annotations is presented in Listing 1. A subset of named parameters lifetimes can be automatically inferred by the process of *lifetime elision* [34]. In short, elision sets a different lifetime for each unspecified parameter and return value lifetime. Afterwards, in the case of only one lifetime existing in both the parameters and return value, they are merged into the same lifetime. If any lifetime in the return value remains unmatched with a lifetime from the parameters, the function definition is declared illegal, as it constitutes either a dangling pointer or a lifetime whose origin is ambiguous.

4.5 Borrow Checker Algorithm

To replicate the borrow checker itself, we largely follow the Non-lexical lifetimes algorithm as described by Matsakis [21], with few to no adaptations. As such, we directly translated the concept of lifetimes (or region variables), the rules for generating the various constraints, the inference algorithm, the in-scope loans, and the final error detection phase. The largest divergence from Rust lies in the handling of some edge cases, such as two-phase borrows [20], due to Rust's desugaring on nested functions or "methods".

5 Implementation

We developed a small proof-of-concept prototype of a C borrow checker to demonstrate the feasibility of a subset of the techniques proposed, using the Clava [3] source-to-source compiler to perform its analysis and transformations. Clava provides a custom intermediate representation (IR) heavily inspired by Clang's [16] AST. More specifically, it uses Clang to parse the source code, and builds its custom IR from Clang's AST, inheriting a similar structure to Clang.

5.1 System Design

On a high-level, it is divided into two steps, first normalization, and then analysis. The normalization step is responsible

for transforming the input program into a form that is easier to analyse, and is applied globally. The analysis step is responsible for performing the actual analysis required to perform the borrow checker algorithm, but is applied at an intraprocedural level.

The C AST differs significantly from that of Rust's, and we had no intention of mimicking the internal structures of the rustc compiler, as we had no guarantees they would adequately translate to C. In spite of that, various components naturally wound up resembling some of the classes and structures used by rustc. The best example of this lies in the representation of paths, as well as the connection between a path, as part of a *value expression*, and the corresponding type it would evaluate to.

Internally, we utilized compilation passes to perform the various analysis required for the borrow checker, allowing for an easy integration with other transformations already provided by Clava. This also aided its development, as it allows for the analysis to be performed in a step-by-step fashion, and for the results of each step to be easily inspected.

5.2 Normalization Stage

Of the various normalization options previously discussed, we have focused on the most crucial ones in order to achieve an MIR-like representation of the input program. In other words, the focus was on eliminating all nested expressions. Following this, we decompose every line of code into a `LVALUE "=" RVALUE` expression, not dissimilar to Rust. This allows further analysis to not have to reason about the order of operations, and instead focus on the individual operations themselves. The normalization also normalizes certain constructs, such as the ternary operator, which is transformed into an `if-else` statement.

Most of the statement decomposition utilized the work of Matos [18], included in the Clava source-to-source compiler [3]. These transformations introduce multiple temporary variables, which are then used to represent the intermediate values of the expression. Every temporary variable has a unique identifier, in the format of `TMP_n`, where n unique identifies the variable. This is important, as it ensures that some later value expressions will only be composed of a single memory location.

The removal of shadowing (re-declaration of a variable with the same name, in an inner scope) was also extremely helpful, since it allowed later analyses to assume that every variable is unique, and thus could be used as an identifier to easily match an expression to the type produced by it.

5.3 Analysis Stage

The analysis stage represents the bulk of the prototype, and is performed as a series of intraprocedural analyses, by iterating through every function definition, and applying a full sequence of analyses and transformations to the respective body. For each, it performs a sequence of seven steps, as present in Figure 1. The order in which they are analysed is not relevant, and, in theory, they could even be performed in parallel.

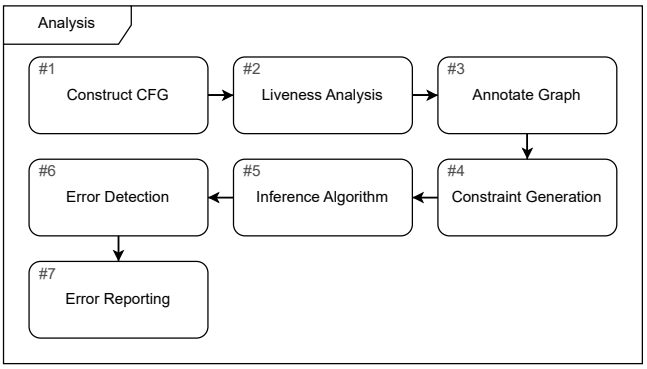

Figure 1. Diagram describing the seven steps of the borrow checker's analysis

In regards to type checking, we mostly rely on Clang to perform it for us, as it is internally used by Clava to generate its own internal intermediate representation. If it used gcc instead, for example, the definition of some extra flags would be required, such as `-Wdiscard-qualifiers`, in order to prevent the creation of a mutable pointer from an object marked as `const`.

To elaborate on each analysis step, we started by constructing a Control Flow Graph (CFG), built internally over a Cytoscape.js [7] graph, a graph theory and networking library, which comes prepackaged with various graph theory algorithms and allows for the simple manipulation of nodes, especially with the usage of the *scratch pad*. The scratch pad is a key-value store associated with each node, and can be used to store arbitrary information. It was extensively used, thanks to allowing for the storage of relevant information directly in the nodes themselves, as well as for the consumption of it in later steps, without the need for recalculations. Moreover, it also allows for multiple objects to only be instantiated once, and then reused throughout the analysis, as is the case for region variables and loans. In turn, this allows for the direct comparison of objects, instead of having to compare their attributes, which allows for the usage of JavaScript's built-in `Set` data structure.

For the second step, a liveness analysis is performed, configured to separate each code block into individual lines of code, resulting in a single unique string to identify each operation in the decomposed function body. For the correctness of NLL, however, drop statements must be disregarded during liveness. Fortunately, by temporarily representing drops as a `#pragma` directive, they are safely ignored by the liveness algorithm.

Thirdly, we annotate each node in the CFG with the data required by later analysis, such as marking "MIR actions"

(i.e. an assignment or function call) for error detection, identifying borrow expressions, or adding typing annotations. In order to avoid unnecessarily iterating over the function body, every annotation is created in a single pass, and then cached into the node itself. This step tackles some of the larger road blocks of implementing a C borrow checker, due to the language lacking much of the required syntax.

To surpass those challenges, we first simulated a pseudo-type system, built on top of the Clang AST's types, with the addition of support for lifetimes in reference types and structs. Moreover, a second hierarchy was created to represent every possible *lvalue* path, this is, every possible expression that represents a valid memory address. This allows for both hierarchies to be interconnected, as when an *lvalue* path is evaluated, it produces a specific type, possibly with lifetimes attached. These paths are then further used to identify loans, as well as to categorize each memory access according to a matrix of *read* or *write*, and *deep* or *shallow*.

The fourth step is responsible for generating the constraints required for the NLL algorithm, as described by Matsakis [21]. Liveness constraints are directly applied to the initial set of each `RegionVar`, and are never instantiated. The remaining outlives constraints are stored in a list, to be utilized later. Of these, subtyping constraints require reasoning adapted to each variable type's variance [28] [29].

The fifth step propagates these constraints across the region variables, or lifetimes, which constitutes a fixed-point iteration problem. In short, each constraint is iterated over and applied until no further changes occur. Each `RegionVar` is composed of its unique *id*, and the set of points that compose its region, expressed as strings to account for the end regions of universal lifetimes (such as named parameters or `static`). Each constraint defines an outlives relation between two `RegionVars` and the point P in which it applies. For example, to propagate a $('a : 'b)\ @\ P$ constraint is to ensure that every point (or statement) reachable from point P that is contained in the set of region variable $'b$ must be included in the set of region variable $'a$. This was implemented as a DFS visitor that added the *id* of any point visited into the region set of $'a$.

The sixth step is responsible for the detection of errors, based on the information computed so far. The error detection algorithm requires another dataflow analysis to calculate the set of loans active in each statement, this is, the *in-scope loans*. To implement it, we have utilized a variation on the iterative worklist algorithm described by Muchnick [24]. In this algorithm, we consume nodes from a queue without duplicate entries (enforced through an auxiliary `Set`), initialized to every node in the CFG. While the queue is not empty, a node is removed from the queue, and its *in-scope loans* are calculated based on the *out-scope loans* of its predecessors. If the *in-scope loans* of the node change, its successors are added to the queue, and the process is repeated until the queue is empty. This algorithm minimizes the number of nodes visited, as it visits exclusively nodes whose *in-scope loans* may have changed. Finally, having calculated the in-scope loans, a final iteration through the CFG is performed, and any in-scope loan that violates an annotated `Access` is detected, according to the rules delineated in the NLL RFC [21]. If such a violation is found, an error is reported, and the analysis is terminated.

The final step reports any errors caught in the previous step in a clear human-readable format. We have opted to report the errors similarly to rustc, by printing the error message, accompanied by the relevant statements, with the error highlighted using the three-point "narrative", similar to the three-act trope in traditional story-telling:

- The point in which the loan is created, B
- The point which might have invalidated the borrow, A
- The point in which its reference was later used, U

In order to provide this "narrative", the three points must ensure that U is reachable from A, and A is reachable from B. In turn, errors can be described as a series of "acts", where first we create the borrow in B, then present the point of error A, and finally, the next use of the reference U. This was considered a significant improvement over previous iterations by the Rust development team [21], which previously only reported the points of error and creation, but not the next use of the reference. This is especially useful for use after free errors, by allowing the programmer to know precisely where the reference was freed, and later used.

Thanks to the structure of our annotations in the CFG, we are able to easily identify which memory access caused the error, as well as the point in which the loan was created. Finding the point in which the borrow is next used, however, is more complicated. Instead of reporting the first usage of the reference after the access, we opted to report the last usage of the borrow. To find it, the built-in methods from Cytoscape.js [7] were used to perform a simple DFS search to find the last point of the CFG, reachable from the statement of the incorrect access, that is still included in the borrow's `RegionVar`. In the case of the "last" usage being inside a conditional branch, and the other branch still lead to more usages, it is still a perfectly acceptable point to report, as in that case, both paths would constitute a violation of the borrow checker.

5.4 Results

From a performance standpoint, apart from the first global normalization pass, every function is processed exactly once. We opted to not use a queue to exclusively visit used functions, as it would force the existence of an entry point to the program. Conversely, processing every definition exactly once allows for the analysis of code without a main (e.g., libraries), as well as allowing the possibility of analysing an isolated selection of functions, helpful for an incremental conversion of existing code. With regards to memory, as

Listing 2. Variation upon one of the main examples presented in the NLL RFC [21]

```
void use(const int *a) {

}

int main() {
    int foo = 1;
    int bar = 2;
    const int *p;

    p = &foo;
    if (2 > 1) {
        use(p);
        foo = 4;
        // Other processing
        p = &bar;
        // More processing
    }
    foo = 8;
    use(p);
    return 0;
}
```

there is no need to preserve the CFG and its annotations between the analysis of different functions, they are deleted between each analysed function, reducing the overall memory required to run the borrow checker.

From the standpoint of system design, we have opted to create a custom simplified intermediate representation, modelled after Rust's MIR, and to perform the analysis over it. This allowed us to leverage existing work on the Clava compiler, as well as to hide the complexity of the Join Points (which are strongly tied to Clang's AST) behind a layer of abstraction. This was expressed as a set of annotations built on top of the CFG built for each function body.

We have tested our prototype against programs inspired by the NLL RFC [21], alongside other small synthetic programs to isolate some edge cases. Because the goal of the borrow checker is to accurately detect and report violations of the ownership and borrowing rules, we have also focused on creating variations of those examples, in which errors were purposefully introduced. One such variation is shown in Listing 2, in which two final assignments to variable the `foo` were added. The first one, inside the `if` block, is legal, as it is after the value borrowed in `p` is no longer in use. The second, however, is illegal, as it is performed while `foo` is still borrowed inside `p`.

Table 1. Set of constraints generated from Listing 2

Region	Points
$'1$	{*id*_11, *id*_15, *id*_17, *id*_18, *id*_6, *id*_7, *id*_8, *id*_9}
$'2$	{}
$'3$	{}
Outlives Constraints	
($'2 : '1$) @ *id*_6	
($'3 : '1$) @ *id*_15	

Table 2. Final lifetimes computed from the constraint set in Table 1

Region	Points
$'1$	{*id*_11, *id*_15, *id*_17, *id*_18, *id*_6, *id*_7, *id*_8, *id*_9}
$'2$	{*id*_11, *id*_17, *id*_18, *id*_6, *id*_7, *id*_8, *id*_9}
$'3$	{*id*_15, *id*_17, *id*_18}

The validation of the results produced by each test program was divided into three parts. First, a graphical representation of the CFG with the cached annotations data directly present was generated, and then confronted against the expected values. This allowed for a simple confirmation of both the normalization stage, as well as the first half of the borrow checker algorithm. Afterwards, we compared the set of constraints generated from the code, such as in Table 1. After confirming the correctness at this stage, we then manually calculate the expected values for each region variable, or in other words, lifetime, and compare them to the program output, such as in Table 2.

Finally, given the importance of producing good error reports in the case of failure, we created an equivalent program in Rust, and compared the errors produced by our prototype against those reported by the rustc compiler. Our errors are comprised of the same three-point "narrative" structure utilized by Rust. We can verify that it correctly identified the assignment to `foo` in `id_18` as the violation of its borrow in `p`. It also correctly identified the last usage of the borrow in `id_15` as the last point in which the borrow is still in use.

Unfortunately, due to the normalization and various transformations applied, the lines indicated in the error do not accurately correspond to those of the original source code file. To accurate report the original location, altering Clava would be required. The incorrect line numbers are, however, a minor issue, as the error is still correctly reported. Furthermore, the user has access to the normalized code, in which the line numbers are correct, facilitating the process of an error back to the original source code.

```
error[E0506]: cannot assign to `foo` because it is borrowed
  --> src/main.rs:16:5
   |
8  |     p = &foo;
   |         ---- `foo` is borrowed here
...
16 |     foo = 8;
   |     ^^^^^^^ `foo` is assigned to here but it was already borrowed
17 |     utilize(p);
   |             - borrow later used here

For more information about this error, try `rustc --explain E0506`.
```

Figure 2. Error produced by the rustc compiler for a program equivalent to Listing 2, via https://play.rust-lang.org

```
error[E0506]: Cannot write to 'foo' while borrowed
 ---> nll_used_while_borrowed.c:18
   |
10 |   p = &foo;
   |           (shared) borrow of 'foo' occurs here
18 |   foo = 8;
   |           write to 'foo' occurs here, while borrow is still active
19 |   use(p);
   |           borrow is later used here
   |
```

Figure 3. Error produced by the prototype for Listing 2

6 Conclusions

Every year over the last decade, CVE vulnerabilities associated with memory safety are a significant portion of newly discovered problems with C and C++[32], as developers inadvertently insert memory corruption bugs. This is a clear indication that the current approaches to memory safety are not sufficient, and that new approaches are required. Simultaneously, we have seen the rise of languages promoting memory safety without compromising their runtime performance as their flagship, namely Rust.

In this work, we have identified another avenue to improve the memory safety of C: the application of the ownership and borrowing model that form the core of Rust. To that effect, we identified the key algorithms behind the borrow checker, and proposed methods to retrofit them into C through source-to-source compilation. This resulted in a system capable of enforcing comparable ownership and aliasing rules in C source code, whilst ensuring no compiler lock-in and simplifying any verification of a program's correctness. The key feature of this system is automatic memory management, especially for struct types, achieved through the use of the RAII paradigm, the application of Drop Elaboration to ensure every resource is freed once-and-only-once, as well as the checks of a custom borrow checker that ensures no concurrent mutable memory operations occur, or that a dangling pointer is never dereferenced.

We have implemented a prototype for a subset of the analysis and transformations required for this system, which serves as a proof of concept of the core ideas presented in this work. Its main goal is to replicate the Non-Lexical Lifetimes-based borrow checker, an intraprocedural analysis that enforces the ownership and aliasing rules, with an emphasis on the calculation of lifetimes (or region variables). The prototype is implemented as a set of scripts for the Clava source-to-source compiler, meeting the requirements of a compiler-agnostic solution, while also being easily integrated into existing build environments, such as cmake.

We have evaluated the prototype by writing equivalent programs in C and Rust, and comparing its error reporting to that of the latest version of the Rust compiler. The results showed that it is able to accurately detect the same intended class of errors, and that the error messages produced were also similar in nature, albeit less refined, due to some information loss in parts of the source-to-source compiler's normalizations. This is a clear indication that the core ideas presented in this work are sound, and that with further investigation, could be further developed into a more viable and feature-complete tool for improving the safety of C code.

6.1 Future Work

We believe we have laid the groundwork for a more complete and thorough C borrow checker. However, in order to design a comprehensive borrow checker ready to be integrated into the compilation pipeline of a true project, various issues need to be tackled.

Firstly, this proposal still lacks a comprehensive solution for the handling of arrays, with a focus on indexing and slicing operations, as well as its initialization and respective calls to the allocator. Alongside this support, we also wish to study the addition of runtime checks that also ensure spatial safety, such as bounds checking.

The second largest challenge lies on the integration of calls to the standard library, as well as other third-party C libraries. We believe supporting the standard library could be realistically achieved by establishing a "contract" similar to the boundary between safe and unsafe Rust, as well as through wrappers suitable to analyses pertaining to ownership. Support for third-party C libraries, especially if distributed as pre-compiled binaries, could prove to be particularly problematic, since our proposal relies exclusively on an annotated version of a project's source code. Ultimately, it would allow for the expansion of our test suite to include real-world applications and well-regarded benchmarks.

Further testing of various compilers is also required in order to evaluate the consequences (if any) of using the `restrict` qualifier to identify mutable borrows, mainly due to the relatively unclear nature of the C99 standard on this topic. Furthermore, compilers are free to ignore this qualifier when optimizing the generated binaries.

Finally, we also expect some improvements towards more accurate tracking of the location of an error on the original source, as well as the extension of the prototype to gradually cover more complex cases, mainly revolving around drop elaboration, structs, and incomplete types.

References

[1] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In *2008 IEEE Symposium on Security and Privacy (Sp 2008)*. IEEE, Oakland, CA, USA, 263–277. https://doi.org/10.1109/SP.2008.30

[2] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

[3] João Bispo and João M. P. Cardoso. 2020. Clava: C/C++ Source-to-Source Compilation Using LARA. *SoftwareX* 12 (2020), 100565. https://doi.org/10.1016/j.softx.2020.100565

[4] Robert DeLine. 2001. Enforcing High-Level Protocols in Low-Level Software. *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (May 2001), 59–69.

[5] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 103–114. https://doi.org/10.1145/1346281.1346295

[6] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*. IEEE, Cambridge, MA, 53–60. https://doi.org/10.1109/SecDev.2018.00015

[7] Max Franz, Christian T. Lopes, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D. Bader. 2015. Cytoscape.Js: A Graph Theory Library for Visualisation and Analysis. *Bioinformatics (Oxford, England)* 32, 2 (Sept. 2015), 309–311. https://doi.org/10.1093/bioinformatics/btv557 arXiv:https://academic.oup.com/bioinformatics/article-pdf/32/2/309/49016536/bioinformatics_32_2_309.pdf

[8] Dan Gohman. 2015. *Incorrect Liveness in DeadStoreElimination*. Technical Report 25422. LLVM bugs. https://bugs.llvm.org/show_bug.cgi?id=25422

[9] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 282–293. https://doi.org/10.1145/512529.512563

[10] Reed Hastings and Bob Joyce. 1992. Purify: Fast Detection of Memory Leaks and Access Errors. *Proceedings of the Winter 1992 USENIX Conference* (1992), 125–138.

[11] ISO. 1999. ISO/IEC 9899:1999 - Programming Languages - C.

[12] Ralf Jung. 2023. From Stacks to Trees: A New Aliasing Model for Rust.

[13] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked Borrows: An Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (Dec. 2019). https://doi.org/10.1145/3371109

[14] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language*. No Starch Press.

[15] Felix Klock II. 2014. RFC 0320: Nonzeroing-Dynamic-Drop. https://github.com/rust-lang/rfcs/blob/abc967a2c5ddd0af2d3506897be7ecfbc0e78e97/text/0320-nonzeroing-dynamic-drop.md

[16] Chris Lattner. 2011. LLVM and Clang: Advancing Compiler Technology.

[17] Michael Ling, Yijun Yu, Haitao Wu, Yuan Wang, James R. Cordy, and Ahmed E. Hassan. 2022. In Rust We Trust – A Transpiler from Unsafe C to Safer Rust. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, Pittsburgh, PA, USA, 354–355. https://doi.org/10.1109/ICSE-Companion55297.2022.9793767

[18] João Matos. 2022. *Automatic C/C++ Source-Code Analysis and Normalization*. Ph. D. Dissertation. Universidade do Porto.

[19] Niko Matsakis. 2015. RFC 1211: Mir. https://github.com/rust-lang/rfcs/blob/debadbae2c7fc6cf2d94aef61c08f60b2e6ed297/text/1211-mir.md

[20] Niko Matsakis. 2017. RFC 2025: Nested-Method-Calls. https://github.com/rust-lang/rfcs/blob/188cc17ad38b201867955fb4a51c306c0704b6cf/text/2025-nested-method-calls.md

[21] Niko Matsakis. 2017. RFC 2094: Nll. https://github.com/rust-lang/rfcs/blob/abc967a2c5ddd0af2d3506897be7ecfbc0e78e97/text/2094-nll.md

[22] Niko Matsakis. 2019. Polonius and Region Errors. https://smallcultfollowing.com/babysteps/blog/2019/01/17/polonius-and-region-errors/. (accessed 2023-09-20).

[23] Niko Matsakis. 22. Non-Lexical Lifetimes (NLL) Fully Stable. https://blog.rust-lang.org/2022/08/05/nll-by-default.html. (accessed 2023-08-11).

[24] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[25] Santosh Nagarakatte, Jianzhou Zhao, Milo M K Martin, and Steve Zdancewic. 2009. *SoftBound: Highly Compatible and Complete Spatial Memory Safety for c*. Technical Report MS-CIS-09-01. University of Pennsylvania Department of Computer and Information Science Technical.

[26] George C Necula, Scott McPeak, and Westley Weimer. 2002. CCured Type-Safe Retrofitting of Legacy Code. *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Jan. 2002), 128–139.

[27] Nikita Popov. 2018. Loop Unrolling Incorrectly Duplicates Noalias Metadata. RFC 9405. https://bugs.llvm.org/show_bug.cgi?id=39282

[28] Rust Community. 2014. The Rust Language Reference. https://github.com/rust-lang/reference/tree/effbdc1b059fde09027925e1bea90bb1860d5f27. (accessed 2023-09-05).

[29] Rust Community. 2015. The Rustonomicon. https://github.com/rust-lang/nomicon/tree/302b995bcb24b70fd883980fd174738c3a10b705. (accessed 2023-08-03).

[30] Rust Community. 2018. Polonius Book. https://github.com/rust-lang/polonius/tree/0a754a9e1916c0e7d9ba23668ea33249c7a7b59e. (accessed 2023-09-13).

[31] L. Szekeres, M. Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, Berkeley, CA, 48–62. https://doi.org/10.1109/SP.2013.13

[32] Gavin Thomas. 2019. A Proactive Approach to More Secure Code.

[33] Jim Trevor, Greg Morrisett, James Cheney, Dan Grossman, Michael Hicks, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*. USENIX Association.

[34] Aaron Turon. 2014. *RFC 0141: Lifetime-Elision*. Technical Report 0738. Rust Foundation.

Received 2024-02-29; accepted 2024-04-01