TÉCNICO LISBOA

Static Analysis for C++ Rust-Like Lifetime Annotations

Susana Moreno Monteiro

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Nuno Lopes
Eng. Dmytro Hrybenko

Examination Committee

Chairperson: Prof. Andreas Wichert
Supervisor: Prof. Nuno Lopes
Member of the Committee: Prof. José Fragoso Santos

November 2023

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

Firstly, I would like to express my gratitude to my family, for their unwavering support and encouragement. A special acknowledgment to my parents, Domitília e José, my brother, Francisco, and my boyfriend, Duarte. Thank you for showing interest in my work, for your patience in listening to me talk about it endlessly, and for always being there. Your belief in me and my abilities has been a constant source of motivation. I am truly grateful for your presence in my life.

In addition, I extend my sincere thanks to my dissertation supervisors, Prof. Nuno Lopes and Eng. Dmytro Hrybenko, for their expertise, mentorship, and dedication. Your insightful feedback and valuable input were essential to make this thesis possible. I am also thankful to Google and the engineers Martin Bræne, Luca Versari, Felipe Pereira, and Marcel Hlopko, for their availability, contributions, and remarkable insights.

Last but not least, I want to express my appreciation to my friends and colleagues. Your friendship and the experiences we shared have made this journey enjoyable and memorable.

This thesis would not have been possible without the collective support of these individuals. Thank you for being an integral part of my academic journey.

Abstract

Memory safety vulnerabilities are still a major concern nowadays, being the cause of most security bugs in software. Some programming languages, such as C and C++, are not memory-safe and thus are prone to vulnerabilities introduced inadvertently by developers. There are multiple approaches to mitigate these issues. One of them is using a memory-safe language such as Rust. However, it is not feasible to completely rewrite existing C/C++ code to Rust overnight. Therefore, there is a need for Rust and C/C++ to interoperate. There are specific challenges in designing ergonomic C/C++ APIs for Rust that must be addressed to achieve this interoperability. One of these challenges is the concept of lifetime annotations, which are unique to Rust's memory model and did not exist in C/C++ until recently. If these annotations are incorrectly written, they may invalidate any memory safety guarantees for the entire program, and hence it is essential to get them right. In this thesis, we developed LSA, a tool that uses static analysis to validate lifetime annotations in C and C++ code, thus contributing to the safe and ergonomic interoperability between C/C++ and Rust.

Keywords

Memory safety; Lifetimes; Rust; C; C++; Static analysis.

Resumo

As vulnerabilidades de memória continuam a ser de extrema importância, constituindo a principal causa de erros em software. Algumas linguagens de programação, como C e C++, não garantem segurança de memória e, por isso, são suscetíveis a vulnerabilidades que os programadores possam introduzir inadvertidamente. Existem diversas abordagens para o problema em questão. Uma delas é a utilização de uma linguagem de programação segura em termos de memória, nomeadamente Rust. No entanto, reescrever integralmente programas em C/C++ para Rust é um processo extremamente demorado, pelo que é necessário investir na interoperabilidade entre Rust e C/C++. Esta interoperabilidade envolve vários desafios que têm de ser superados. Um deles é o conceito de anotações de *lifetime*, fundamentais em Rust mas que, até recentemente, não existiam em C/C++. Se estas anotações forem incorretamente introduzidas no código, podem comprometer as garantias de segurança de memória do programa, pelo que é essencial verificar a sua correção. Nesta tese, desenvolvemos a ferramenta LSA, que utiliza análise estática para validar anotações de *lifetime* em C e C++, contribuindo para uma interoperabilidade segura e ergonómica entre C/C++ e Rust.

Palavras Chave

Segurança de memória; Lifetimes; Rust; C; C++; Análise estática.

Contents

List of Figures

List of Tables

Acronyms

AST	Abstract Syntax Tree
CSA	Clang Static Analyzer
CFG	Control-Flow Graph
LSA	Lifetimes Static Analyzer

1

Introduction

Contents

Most security vulnerabilities today are related to memory safety. This is increasingly being acknowledged by many prominent entities, such as the NSA, which alerts for the severity of memory management issues, still commonly exploited today.[1] In order to prevent memory vulnerabilities, the NSA urges organizations to use memory-safe programming languages and, on top of that, use additional memory protection tools and techniques.

According to Microsoft, in 2019, 70% of all security bugs were due to memory safety issues [2]. Similarly, in the Chromium project, which is mostly written in C++, 70% of high-severity security bugs are related to memory[2] and, in 2019, 76% of vulnerabilities found in Android were also attributed to memory safety issues [3, 4]. This number dropped to 35% in 2022, coinciding with the increased adoption of memory-safe programming languages in Android, such as Java, Kotlin, and Rust. For the first time, in 2022, the new code added to Android 13 is mostly written in memory-safe programming languages.

[1] https://tinyurl.com/ms6uncsm
[2] https://www.chromium.org/Home/chromium-security/memory-safety/

Memory safety bugs can be divided into spatial and temporal errors. Spatial errors happen when the program reads from or writes to a memory region outside an allocated object. On the other hand, temporal errors occur when the program accesses an object before it is initialized or after it is deleted.

The C and C++ programming languages are not memory-safe, both allowing spatial and temporal errors. Thus, vulnerabilities introduced inadvertently by developers are quite frequent. By exploiting these vulnerabilities, attackers may gain access to sensitive data or execute arbitrary code, for instance.

To mitigate these vulnerabilities, developers can follow the C++ core guidelines,[3] which provide a set of best practices for writing safe and reliable C++ code. By following these guidelines, developers can improve the quality and reliability of their code, consequently reducing the likelihood of introducing bugs. Nonetheless, following these guidelines does not eliminate the possibility of errors occurring.

Another option to address memory errors is to use specialized tools. There are various tools available that aim to make C/C++ programs safer by detecting or eliminating bugs. Some existing tools perform static analysis, meaning that they detect bugs by analyzing the source code, while others perform dynamic analysis, which involves analyzing the program's behavior while it is running. For instance, code instrumentation tools, such as control-flow integrity tools, dynamically detect certain memory problems and mitigate them. The existing memory protection tools differ greatly in their approaches, and many have limitations that make them unsuitable for use in production environments. Some detect only a few kinds of bugs, while others attempt to be more comprehensive but may consequently sacrifice performance and accuracy. In general, tools that have high performance overhead, low comprehensiveness, or low accuracy are unlikely to be adopted due to their impracticality.

Alternatively, there are memory-safe programming languages, which prevent memory vulnerabilities. Most memory-safe languages, such as Java, C# and Go, have garbage collection mechanisms to automatically deallocate unused memory. The garbage collector ensures temporal memory safety, relieving developers from manual memory management. Using memory-safe languages, instead of C or C++, would provide memory safety to programs and thus eliminate memory vulnerabilities. In fact, memory-safe languages are becoming increasingly popular in diverse areas.

Nevertheless, C and C++ are still widely used because they provide features that most memory-safe, high-level languages do not. C and C++ provide low-level control of memory layout and a proximity to the underlying hardware that safe languages generally cannot offer. Additionally, C/C++ require minimal runtime support and do not have a garbage collector. The garbage collector causes unpredictable pauses in the execution of the program, which may prevent the program from responding to events promptly, as may be necessary in certain domains.

Many of the C/C++ features described above provide the programmer with much more control and make C/C++ significantly more efficient than memory-safe, high-level programming languages. While

[3] https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md

these features are not crucial for many programs, some require the performance and level of control that C/C++ offer. This is the case with operating systems, which need to directly access hardware and a lower-level control. Also, operating systems are not usually implemented in a language that uses garbage collection, which is how most safe languages manage memory. For this reason, C and C++ continue to be widely used in a variety of areas. They are often still the main languages for developing operating systems, virtual machine monitors, language run times, database management systems, embedded software, web browsers (such as Chrome), and more.

The Rust programming language[4] is a strong alternative to C and C++, both in the features it offers and in its use cases. Rust is a memory-safe programming language that was created to ensure high performance and support for concurrent programming, with an emphasis on code safety. Furthermore, it does not have a garbage collector. Instead, Rust has an efficient and strong memory model that provides equivalent temporal memory safety guarantees. In this memory management model, each value has exactly one owner, which determines when the value should be deallocated. The Rust compiler uses information about the owner to enforce temporal memory safety while providing useful error messages.

The popularity of Rust is increasing in multiple fields, including some that are commonly associated with the use of C/C++. Many companies are using Rust in production, in distinct application domains.[5] Some operating systems are already compatible with the development of components in Rust. It is the case of Linux [5], which was implemented originally in C for efficiency. Recently, Rust has been introduced in kernel development to prevent memory and synchronization bugs. In Android, new native OS components contain a considerable percentage of code written in Rust[6], making up to 21% of new code added to Android 13. According to Google, since Rust's introduction in Android, no memory safety vulnerabilities have been found in Android's Rust code [3].

Rust is also being used in web browsers, such as Firefox, which is developing a project called Oxidation[7] to integrate Rust in and around Firefox. In the Chromium project, engineers are also experimenting with Rust.[8]

Another technology where Rust is popular is blockchain. There are some blockchain projects based on Rust, like Solana[9] and Polkadot.[10] The Move language,[11] used for writing smart contracts for the Diem blockchain, is heavily inspired by the Rust programming language. Finally, Rust is used to develop cybersecurity products. For instance, Cloudflare uses Rust to build many of its products, which are required to be secure, efficient, and reliable.[12]

[4] https://doc.rust-lang.org/book/
[5] https://kerkour.com/rust-in-production-2021
[6] https://source.android.com/docs/setup/build/rust/building-rust-modules/overview
[7] https://wiki.mozilla.org/Oxidation
[8] https://www.chromium.org/Home/chromium-security/memorysafety/rust-and-c-interoperability/
[9] https://solana.com
[10] https://polkadot.network
[11] https://move-book.com, https://github.com/diem/move
[12] https://blog.cloudflare.com/tag/rust/

Since Rust is a strong alternative to C/C++ in many situations, translating code from C/C++ to Rust can be an excellent approach to address memory vulnerabilities in C/C++ code. There are more and more projects migrating from C and C++ to Rust, and there are already several ongoing efforts to solve the problem of translating C/C++ code into Rust [6–10]. These tools aim to automate much of the translation from C/C++ to Rust, thus making the migration easier, practical, and scalable. However, translating C/C++ code to Rust is not trivial. Firstly, there are millions of lines of code written in C and C++, making it impossible to transition completely from C/C++ to a safe language overnight. Therefore, we need Rust and C/C++ to interoperate so that this translation can be performed incrementally.

Secondly, in order to automatically translate a program from an unsafe programming language to a safe one, it is necessary to prove beforehand that the program is safe memory-wise. This is not always possible due to technological and theoretical limitations. Besides, supposing that proving that a program was completely safe memory-wise was possible, then it would not be necessary to translate it to a memory-safe language.

Finally, there are several specific challenges in designing ergonomic C/C++ APIs for Rust programs. These obstacles prevent the full replacement of C/C++ programs and therefore it is crucial to address them. One of these challenges is related to Rust's unique memory model. In Rust, each value has a single owner and each reference a lifetime, which represents the scope for which that reference is valid.

However, these concepts do not exist in C and C++, so when trying to interoperate C/C++ and Rust, the owners and lifetimes must be inferred in some way. One option would be to use static analysis to compute this, but it is not always possible. Another solution is to introduce lifetime annotations in C and C++ code, although, just like in Rust, it might be burdensome for the developers because it requires them to manually specify the lifetimes.

1.1 Contributions

This thesis revolves around the interoperability between the C/C++ and Rust programming languages. Particularly, it focuses on a specific challenge to this interoperability: Rust-like lifetime annotations. In Rust, each reference has a lifetime and the Rust compiler uses these lifetimes to keep track of the validity of each reference. This way, it can prevent dangling pointers and other memory errors. Lifetimes form the basis of Rust's memory model, which ensures memory safety without requiring a garbage collector.

The concept of lifetimes is specific to Rust and thus did not exist in C/C++. However, Rust-like lifetimes annotations were recently implemented in Crubit,[13] a C++/Rust interoperability tool, and proposed to be included in Clang[14] as an extension to C++ [11].

[13] https://github.com/google/crubit
[14] https://clang.llvm.org/

Below we show an example of these C++ Rust-like lifetime annotations.[15] The function `smallest`, written in C++, receives two string references and returns the shortest of the two. In this case, the return value should be valid as long as the function's arguments are, which can be expressed by annotating all of them with the same lifetime `$a`.

```
1  const std::string& $a smallest(
2      const std::string& $a s1,
3      const std::string& $a s2) {
4    if (s1.length() < s2.length()) {
5      return s1;
6    } else {
7      return s2;
8    }
9  }
```

Crubit includes a tool that automatically infers these C++ Rust-like lifetimes annotations. The inferred annotations can then be mapped to Rust lifetimes, contributing to Rust and C++ interoperability. However, it is still necessary to create a tool that verifies the correctness of C++ lifetime annotations. This tool is crucial for ensuring the correctness of C++ lifetime annotations added or modified manually by developers. In addition, it may be used to validate programs automatically annotated by Crubit, as well as verify that a function continues to be lifetime-correct when its implementation is changed.

It is to note that these annotations will be used by translation and interoperability tools between C++ and Rust. These tools rely on the assumption that the C++ lifetime annotations are correct. However, if they are not, the tools may generate invalid code. To avoid this, it is important to have a verification tool that ensures the correctness and reliability of the Rust-like C++ lifetime annotations.

In this thesis, we use static analysis to implement a tool, called Lifetimes Static Analyzer (LSA), that validates Rust-like lifetime annotations in C++ code, according to Rust's lifetime rules, thus contributing to reliable C++/Rust interoperability.

Below we present two examples that illustrate the type of errors that LSA targets:

```
1  int *g;
2
3  void f1(int *x) {
4      int **p = &g;
5      *p = x;
6  }
```

The assignment on the fifth line indirectly modifies the value of `*g`, which is a global variable, due to the previous line. This creates a dangling pointer since, after the end of `f1`, `x` may no longer exist, leaving `g` pointing to an undefined memory location.

The following example illustrates a different type of error that LSA can detect:

[15] This example was taken from https://discourse.llvm.org/t/rfc-lifetime-annotations-for-c/61377

```
1  void f2(int **p) {
2      int i = 1;
3      *p = &i;
4  }
5
6  int* f3(int **x) {
7      f2(x);
8      return *x;
9  }
```

In function `f2`, `*p` is passed to the function by reference, since it is an inner indirection of the parameter `p`. Consequently, any modifications made to `*p` inside `f2` will affect the corresponding argument in `f3`. In this case, inside `f2`, `*p` is assigned the address of the local variable `i`, which is deleted when the function returns, rendering the value of `*p` invalid. LSA can identify these safety problems through the analysis of lifetimes in C and C++, generating a warning in each of these cases.

The following set of characteristics ensures that LSA is effective, practical, and likely to be adopted:

- The analysis is fast, yielding a slowdown of no more than 10% on the tested benchmarks, when compiled with the `-O0` flag.
- The number of false positives is minimal.
- The semantics are based on Rust's lifetime analysis, ensuring consistency in the results of the two analyses.
- The error messages are short and informative, helping the developer in the process of making the necessary corrections to the lifetime annotations.

LSA is implemented on Clang,[16] making use of the recent C++ Rust-like lifetime annotations.

1.2 Outline

This document is structured as follows. In Chapter 2 we introduce the Rust programming language and the concept of lifetimes. Then, in Chapter 3 we introduce some fundamental concepts of static analysis, and in Chapter 4 we discuss tools and techniques related to memory safety. The design and implementation of LSA are described in Chapters 5 and 6, followed by the results and discussion in Chapters 7 and 8.

[16] https://clang.llvm.org/extra/clang-tidy/

2

Rust and Lifetimes

Contents

Rust is a programming language that provides high-performance, type, memory and thread safety, concurrency, high-level ergonomics, and low-level control. In short, it is a memory-safe language that offers features that most memory-safe languages, such as Java and C#, cannot offer. For instance, it offers high performance and low-level control, making it a strong memory-safe alternative to C/C++.

In Section 2.1, we delve into Rust and its memory model, which sets it apart from other programming languages and is also the focus of this thesis. Then, in Section 2.2, we explore the concept of lifetimes.

2.1 Rust

Memory safety can be divided into spatial and temporal memory safety. These two types of memory safety require different approaches to be enforced. Rust ensures spatial memory safety through bound checks at run time, which is a common approach among most memory-safe languages. On the other hand, Rust provides temporal memory safety without using a garbage collector, unlike many other

memory-safe languages such as Java, C#, and Go. Instead, Rust ensures temporal memory safety, at compile time, through the implementation of a specific ownership system.[1] This ownership system is a set of rules used to manage memory and ensure there are no memory vulnerabilities. In this ownership system, each value has exactly one owner and each reference has a specific lifetime. The reference's lifetime represents its scope, determining the part of the program where the reference is valid.[2]

In Rust, an object is only valid while it has an owner. In other words, an object is valid while the variable that holds it is in scope. Thus, when the owner goes out of scope, Rust automatically invalidates that variable and cleans up everything related to the respective object.

However, the developer may want to assign the object to another binding, which requires changing its owner. The following code shows an example of that.[3]

```
1  let v = vec![1, 2, 3];          // v is the owner of the vector
2  let v2 = v;                     // v2 is the new owner of the vector
3  println!("v[0] is: {}", v[0]);  // compile-time error
```

The operation on line 2 does not perform a deep copy of the vector but, instead, moves the ownership of the vector from `v` to `v2`. This invalidates the binding `v` since, as stated previously, each value has one and only one owner in Rust. Thus, reading the value of `v` in line 3 results in a compile-time error.

The ownership concept also applies to values passed as arguments to functions. When a value is passed to a function call, ownership is transferred implicitly from the variable to the function parameter. Consequently, the variable becomes invalid and cannot be used after the function returns.

```
1  fn take(v: Vec<i32>) { /* ... */ }
2
3  let v = vec![1, 2, 3];
4  take(v);
5  println!("v[0] is: {}", v[0]);  // compile-time error
```

It is often useful that the variable remains valid after the function call. This can be achieved by transferring ownership back when the function returns, which is bothersome in many cases. As an alternative, functions can take references to values instead of the values themselves, consequently never receiving ownership of the value. References are pointers that are guaranteed to point to a valid value of a particular type.

In Rust, there are three types of indirections: pointers, references, and smart pointers. Pointers, or "raw" pointers in Rust, are variables that store memory addresses, which then point to other data. They are equivalent in C++ and Rust, being unsafe and nullable in both programming languages. In Rust, pointers cannot be verified by the Rust compiler and do not provide temporal memory safety. Likewise, pointers in Rust may not necessarily point to a valid memory address, resulting in potential violations of spatial memory safety. Additionally, they do not implement any automatic cleanup.

[1] https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html
[2] https://doc.rust-lang.org/rust-by-example/scope/lifetime.html
[3] The examples of Rust in this section were taken from https://doc.rust-lang.org/book/

Since Rust programs are memory-safe and pointers are not memory-safe, these cannot be used in safe Rust code. To use raw pointers, they must be placed inside a block of code annotated with the `unsafe` keyword.[4] The compiler does not verify if the code inside of this block is memory-safe. Therefore, the `unsafe` keyword is an escape hatch that offers developers a set of abilities that are not possible in safe Rust, one of them being the use of pointers.

Unlike pointers, the concept of references[5] is specific to Rust. They are memory-safe, ensuring spatial and temporal safety. Additionally, they are non-nullable and have lifetimes, which allow the Rust compiler to validate their usage. References are much more common in Rust than plain pointers and developers are encouraged to use them since they provide these safety features without any overhead. The scope of references ends on the last time they are used.

Finally, smart pointers[6] are a third kind of indirections in Rust. The concept of smart pointers also exists in C++. There are multiple types of smart pointers in C++, namely `std::unique_ptr`,[7] `std::shared_ptr`,[8] and `std::weak_ptr`. In C++, a smart pointer is a wrapper class over a pointer that automatically frees the memory of the object when there are no more pointers to it.

In Rust, smart pointers are structures that carry metadata and offer specific capabilities, not found in references. The main difference between references and smart pointers is that, while the former only borrow data, the latter can also take ownership of the data they point to. There are multiple types of smart pointers in Rust, each serving a specific purpose. For instance, there are types of smart pointers in Rust equivalent to the C++ `std::unique_ptr` and `std::shared_ptr`. These are the `Box<T>` and `Rc<T>`/`Arc<T>` types, respectively.

Both boxes (`Box<T>`)[9] and the C++ `std::unique_ptr` ensure that the object they point to has only one owner. They allow data to be stored on the heap and automatically free the data when they reach the end of their scope. Boxes are also useful in Rust when the size of the object cannot be known at compile time.

Reference counted pointers (`Rc<T>`),[10] atomically reference counted pointers (`Arc<T>`)[11] and the C++ equivalent `std::shared_ptr` all allow a value to have multiple owners, unlike the Rust references. These pointers keep track of the number of owners so that, when a value has no more owners, the data gets deallocated. The difference between `Rc<T>` and `Arc<T>` is that the latter uses atomic operations and hence it is thread-safe.

Other examples of smart pointers are `String` and `Vec<T>`. Both of them are smart pointers because they own heap data, unlike references, and allow us to manipulate them. There are many other types

[4] https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html
[5] https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html
[6] https://doc.rust-lang.org/book/ch15-00-smart-pointers.html
[7] https://eel.is/c++draft/unique.ptr
[8] https://eel.is/c++draft/util.sharedptr
[9] https://doc.rust-lang.org/book/ch15-01-box.html
[10] https://doc.rust-lang.org/book/ch15-04-rc.html
[11] https://doc.rust-lang.org/std/sync/struct.Arc.html

of smart pointers available. Multiple libraries provide custom smart pointers and developers can also implement their own using structs.

The scope of references and smart pointers differs between C++ and Rust. In C++, a smart pointer goes out of scope when it reaches the end of the block it was created in. When this happens, the object the smart pointer points to is automatically deleted. The exception is `std::shared_ptr`, in which case the counter that keeps track of the number of references to that object is decreased and the object is only deleted when there are no more pointers to it. An example of the scope of smart pointers in C++ is provided below.[12]

```
1  void UseSmartPointer() {
2      auto song = std::make_unique<Song>("Nothing on You", "Bruno Mars");
3      // ...
4  } // song is automatically deleted here
```

Going back to the example of the `take` function, references can be used to fix the error in line 5. In this case, instead of taking the `Vec<i32>` as an argument, the function could take the reference `&Vec<i32>`. Below is a version of the `take` function that uses references, making variable `v` still valid when the function returns. The process of creating a reference to a value is called borrowing. Unlike the previous examples, where the ownership of the vector was transferred, borrowing allows the reference to use the vector without changing the vector's owner.

```
1  fn take(v: &Vec<i32>) { /* ... */ }
2
3  let v = vec![1, 2, 3];
4  take(&v);
5  println!("v[0] is: {}", v[0]);  // no error
```

The concept of borrowing allows a value to have multiple references to it, which could potentially lead to temporal errors such as dangling references and data races. Dangling references are pointers to invalid data or an incorrect location. Data races happen when two or more pointers access the same location at the same time, where at least one of them is writing. However, these issues are prevented by Rust's memory model.

In order to eliminate these problems, there are some rules for borrowing in Rust. Firstly, the owner of a value must last, at least, as long as any of its references, preventing the program from using a reference to access data that is no longer valid. Secondly, there can be multiple immutable references to a value or exactly one mutable reference, but not both at the same time. This way Rust provides concurrency without data races.

In Rust, by default, variables are immutable, meaning that once they are attributed a value, Rust guarantees that it cannot be changed. This helps to prevent unintended modifications to variables and

[12]This example was taken from https://www.internalpointers.com/post/beginner-s-look-smart-pointers-modern-c

thus developers are encouraged to use immutable variables in Rust. However, it can be useful to have mutable variables and Rust provides the keyword `mut` to let developers declare them.

References are also immutable by default and can be declared mutable by using the `&mut` type constructor. In the example above, the reference to `v` is immutable because it was not declared using the `mut` operator. This means that the reference cannot modify the value of `v`. However, in this case, we could declare the reference mutable as it is the only reference to `v`. Thus, we could rewrite the code in the following way to have a mutable reference:

```
1  fn modify(v: &mut Vec<i32>) { /* modifies v */ }
2
3  let mut v = vec![1, 2, 3];
4  modify(&mut v);
5  println!("v[0] is: {}", v[0]);  // no error
```

In line 6, the operator `&mut` is used to create a mutable reference to `v`. This means that the function `modify` can use the reference to change the value of `v`. It is to be noted that, for the reference to be mutable, the variable itself must also be mutable. Consequently, variable `v` must be declared using the `mut` keyword, in line 3.

The Rust compiler has a borrow checker,[13] which is responsible for statically verifying a set of properties to ensure temporal memory safety, at compile time. For instance, it makes sure that all variables are initialized before being used. The borrow checker has to enforce the ownership and borrowing rules of Rust's memory model. This includes ensuring there are never multiple mutable references to the same value and that values that are immutably borrowed cannot be changed. Additionally, the borrow checker verifies that all borrows are valid by ensuring that references never point to data that might have been deallocated. To ensure the correctness of all borrows, the borrow checker needs to know the lifetime of each reference and determine the relationship between references.

2.2 Lifetimes

The main goal of lifetimes is to prevent dangling references, i.e., pointers to an invalid or wrong location. An example of a dangling reference is given below. The lifetime of each variable is described in the respective comment.

```
1  let r;                // Introduce reference: r
2  {
3      let i = 1;        // Introduce scoped value: i
4      r = &i;           // Store reference of i in r
5  }                     // i goes out of scope and is dropped.
6
7  println!("{}", r);   // r still refers to i, compile-time error
```

[13] https://rustc-dev-guide.rust-lang.org/borrow_check.html

Variables `r` and `i` are created in different scopes, thus the lifetime of `r` is different from the lifetime of `i`. Specifically, the lifetime of `i` is shorter than the lifetime of `r`. At compile time, the borrow checker compares the lifetimes, concluding that `r` refers to an object with a shorter lifetime.

This example breaks one of the borrowing rules: the owner of a value (in this case, `i`) must last longer than any of its references (in this case, `r`). Using lifetimes, the borrow checker can report the above issue and hence protect Rust's programs against temporal errors. In contrast, C++ compilers accept this code, which triggers undefined behavior. Therefore, the C++ compiler is free to optimize the code in whatever way it wants (including removing the whole code) or let it crash at run time, for example.

Rust's borrow checker automatically attributes default lifetimes in simple cases. However, in more complex situations, Rust requires developers to add explicit lifetime annotations to the code. These annotations provide the borrow checker with information about how references relate to each other, helping it to correctly verify the code.

2.2.1 Lifetime annotation syntax

In Rust, lifetime annotations take the form of an apostrophe (`'`) followed by the name associated with the lifetime. Here is an example of a simple reference and a second one annotated with lifetime `'a`.

```
1  &i32        // simple reference
2  &'a i32     // reference annotated with explicit lifetime
```

Lifetime annotations can also be used in function signatures to specify the lifetimes of function parameters and the return value.

```
fn foo<'a, 'b>(x: &'a i32, y: &'a i32, z: &'b i32) -> &'a str { /* ... */ }
```

The lifetime `'a` is declared as a generic parameter and is associated with both `x`, `y`, and the return value of `foo`. This means that `'a` represents the shortest lifetime among them. On the other hand, `z` is annotated with a different lifetime, `'b`, and therefore its lifetime is not related to the others.

It should be noted that a lifetime annotation by itself does not specify the lifetime of a single variable. Instead, these annotations express how the lifetimes of different references are related to each other. The exception is `'static`, which is a special lifetime that denotes that the affected value lives for the entire duration of the program. It is the largest lifetime and is attributed to global variables, for instance, since they exist during the whole program.

As stated above, every reference has a lifetime associated with it. However, in some cases, the borrow checker does not need lifetime annotations. This is called lifetime elision and it is possible because, in specific situations, lifetimes always follow the same pattern and thus are predictable, obviating the

need for explicit annotations.[14] Lifetime elision enables developers to save time and effort because they do not have to manually specify the lifetime of every value.

Below is an example of a function that compiles with no need for lifetime annotations. The function `first_word` receives a reference to a string and returns the first word of this string.

```
fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }
    &s[..]
}
```

The function `first_word` could also be written with explicit lifetimes, in the following way:

```
fn first_word<'a>(s: &'a str) -> &'a str { /* ... */ }
```

In more complex cases, lifetimes do not follow such predictable patterns and thus, the developer is required to add explicit lifetime annotations.

It is to note that lifetime annotations do not change how long the references are valid for. They are merely descriptive and simply give information to the compiler to check if the references' lifetimes are valid when the compiler cannot infer them. Below we show the function `smallest`, which requires explicit lifetimes. This function receives two string references and returns the shortest of the two.

```
fn smallest(x: &str, y: &str) -> &str {
    if x.len() < y.len() {
        x
    } else {
        y
    }
}
```

In this case, Rust's borrow checker does not know which will be the reference returned at compile time because it depends on the arguments passed to it. The borrow checker requires lifetime annotations to learn the relationship between the references' lifetimes and, thus, determine if the reference returned will be valid. This is because the borrow checker only performs an intra-procedural analysis, which means that it only analyzes the control flow inside a single function. Therefore, it does not consider how data is used outside and passed to the given function. By performing only an intra-procedural analysis, the borrow checker can be faster and more practical, but on the other hand, places on the developer the responsibility of annotating the code with lifetimes.

[14] https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#lifetime-elision

The lifetime of the reference returned must match the lifetime of one of the arguments. For the function `smallest` to compile, the developer needs to provide explicit lifetime annotations for the parameters and return reference. In this case, the return value should be valid as long as the function's arguments are. The developer expresses this constraint by annotating both parameters and the return value with the same lifetime, as displayed below. By using the lifetime annotations, the borrow checker ensures that the returned reference is valid and safe to use.

```
fn smallest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() < y.len() {
        x
    } else {
        y
    }
}
```

Lifetime annotations are also useful in structs and vectors. Below is an example where lifetime annotations are required for both of them.

```
struct Item<'a> {
    name: &'a str,
    avg_rating: u32,
}

struct Shelf<'a> {
    name: &'a str,
    items: Vec<&'a Item<'a>>,
}

fn main() {
    let item1 = Item { name: "Book 1", avg_rating: 4 };
    let item2 = Item { name: "Book 2", avg_rating: 5 };
    let shelf = Shelf {
        name: "Shelf",
        items: vec![&item1, &item2]
    };
}
```

Lifetime annotations are needed in this case because the `Shelf` struct contains a reference to a string slice and a vector of references to `Item` structs. `Item`, in turn, also contains a reference to a string slice and thus also needs to be annotated. Lifetimes always need to be specified within struct bodies, and thus compiling the program above would result in an error.

2.2.2 Rust-like lifetime annotations for C++

As mentioned in Chapter 1, the concept of annotating variables with lifetimes did not originally exist in C/C++. However, Rust-like lifetime annotations were recently implemented in Crubit, using Clang's

`clang::annotate_type` attribute.[15] This addition was intended to improve C++/Rust interoperability and static analysis capabilities, amongst other goals.

The current implementation of C++ lifetime annotations includes a tool that infers lifetime annotations from existing C++ code (Crubit) but does not yet include a verification tool. The automatically inferred C++ lifetime annotations can be mapped to Rust lifetimes and, consequently, help map C++ pointers to safe references in Rust. They also improve the code readability for humans, meaning that developers can trust this information to be correct and make use of it when reading the code.

We present some examples of C++ programs annotated with lifetimes. Recall the function `smallest`, written in Rust. This function is rewritten in C++ below, with the respective lifetimes.[16] The lifetimes are declared by the annotation `$a`.

```
const std::string& $a smallest(
    const std::string& $a s1,
    const std::string& $a s2) {
  if (s1.size() < s2.size()) {
    return s1;
  } else {
    return s2;
  }
}
```

In a `class`, the lifetime of pointer `this` is placed at the end of its functions declarations. In the example below, the use of the lifetime annotation `$a` expresses that the lifetime of the value returned by the function `smallest` is equal to the lifetime of the `StringPair` object.

```
class StringPair {
  std::string first, second;
  const std::string& $a smallest() const $a {
    if (first.length() < second.length()) {
      return first;
    } else {
      return second;
    }
  }
};
```

Similarly to Rust, there is also lifetime `$static` in C/C++, which is attributed to global variables, for instance. In C++, the lifetime of smart pointers, such as `std::unique_ptr`, is defined by the scope they were created in. Finally, there is no default lifetime for pointers allocated on the heap, and the reason behind this is discussed in Section 8.6.

It is important to note that the addition of lifetime annotations can improve temporal memory safety, but it has no impact on spatial memory safety. Additionally, the implementation of Rust-style annotations

[15] https://discourse.llvm.org/t/rfc-new-attribute-annotate-type-iteration-2/61378
[16] C++ examples in this section were taken from https://discourse.llvm.org/t/rfc-lifetime-annotations-for-c/61377

in C++ is not the same as implementing a Rust-like borrow checker and thus is not enough to provide complete temporal memory safety.

In Rust, a variable can only have either multiple immutable references or a single mutable reference.[17] At this point, we consider it impractical to apply Rust-like borrow checker rules to C++, because they are often violated in existing code. However, further research might find a way to bring this idea to C++.

This C++ implementation of Rust-like lifetime annotations has some limitations compared to the original lifetime annotations. For instance, it is not possible to specify that a pointer should outlive another. In the following C++ example, we want to specifically express that `b` should live at least as long as the elements of `a`, but this is not possible with the current annotations.

```
void push_first(std::vector<int*>& x, std::vector<int*>& y) {
  x.push_back(y[0]);
}
```

In Rust, it is possible to represent this constraint using a `where` clause, which allows developers to specify constraints on generic parameters and lifetimes.[18] The `push_first` function could be written in Rust in the following way, using a `where` clause.

```
fn push_first<'a, 'b>(x: &mut Vec<&'a i32>, y: &mut Vec<&'b i32>)
  where 'b : 'a
{
    x.push(y[0]);
}
```

Here, the `where` clause specifies that the lifetime `'b` must outlive `'a`. This ensures that the value pushed inside the vector `x` will not outlive the vector itself. If this constraint is violated, the code will not compile.

Nevertheless, this implementation of C++ Rust-like lifetime annotation covers most situations where lifetime annotations are needed.

[17] https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html#the-rules-of-references
[18] https://doc.rust-lang.org/std/keyword.where.html

3

Static Program Analysis

Contents

Static program analysis [12] is a technique that involves inspecting program code and understanding its behavior without actually executing it. This approach aims to consider all program courses that might occur at run time. Consequently, it not only detects errors in programs but also aims to provide guarantees about a program's behavior by checking all its potential executions, when possible. Static analysis is useful in a range of applications, including compiler optimizations, automatic error detection, and verification of program correctness.

Static analysis works by creating an abstraction model of the state of the program (i.e., a valuation of the program variables and the memory locations) and then determining how the program responds to it. In order to consider multiple execution paths, the analysis must keep track of multiple different possible states. This is usually unreasonable since state space rapidly grows due to branches in the code. There can also be an arbitrary number of different user inputs, which may have to be represented by different states. To address this, static analysis involves creating an abstract model of the program states, which allows the analysis to store less information while sacrificing some level of precision.

Typically, static analysis tools strive for soundness and completeness, although attaining both simultaneously is unrealistic. Soundness ensures that the results of the analysis accurately describe the program's behavior, regardless of the input or the environment. Nevertheless, this does not imply that the analysis can always prove the correctness of a program, which is known as completeness. A verification tool is complete if it never misses any of the errors designed to detect, even if it sometimes produces spurious warnings. While some static analysis tools can provide completeness in simple cases, proving a program's correctness is not always feasible due to technological and theoretical limitations. In fact, typically static analysis tools are used on undecidable properties, which means that, even in theory, it is not always possible to determine if the property holds.

In contrast, dynamic analysis involves running a program and inspecting its output, thus focusing on the current execution path. For this reason, while dynamic analysis can identify errors, it generally cannot prove their absence and the results of a dynamic analysis of a program may not apply to future executions. Therefore, the main challenge of performing a good dynamic analysis lies in selecting a comprehensive set of test cases.

Nevertheless, depending on the problem, it may be possible to design a dynamic analysis that uses very precise approximations. Additionally, for some problems, dynamic analysis can also have low overhead, being almost as fast as program execution. On the other hand, achieving precise results through static analysis requires considerable computational effort and introduces a significant performance overhead.

Static analysis tools can be classified based on a variety of properties, including flow-sensitivity, context-sensitivity, field-sensitivity, and false positives/negatives. In this chapter, we delve into these characteristics and then present the key challenges of static program analysis. However, before doing so, we begin by introducing the concept of type systems and defining a simple language and analysis. These will also be useful to explain the mentioned static analysis characteristics. Additionally, we briefly introduce common useful data structures to represent a program's code when performing static analysis.

3.1 Program representation

To perform static analysis of a program, it is often useful to represent its code in different structures. The most popular ones are Abstract Syntax Trees (ASTs) and Control-Flow Graphs (CFGs). We use the following C code snippet to help highlight the differences between them.

```
1  int fn() {
2      int x = 10;
3      while (x > 0) {
4          x = x - 2;
5      }
6      return x;
7  }
```

An AST (Figure 3.1) is a representation of a program's source code in a tree structure, highlighting its hierarchical organization. Each node in the tree structure corresponds to a statement or an expression and they are connected by edges that represent the hierarchical relationships between them. A statement in a program is a line of code that performs a specific task, such as variable declarations or assignments.

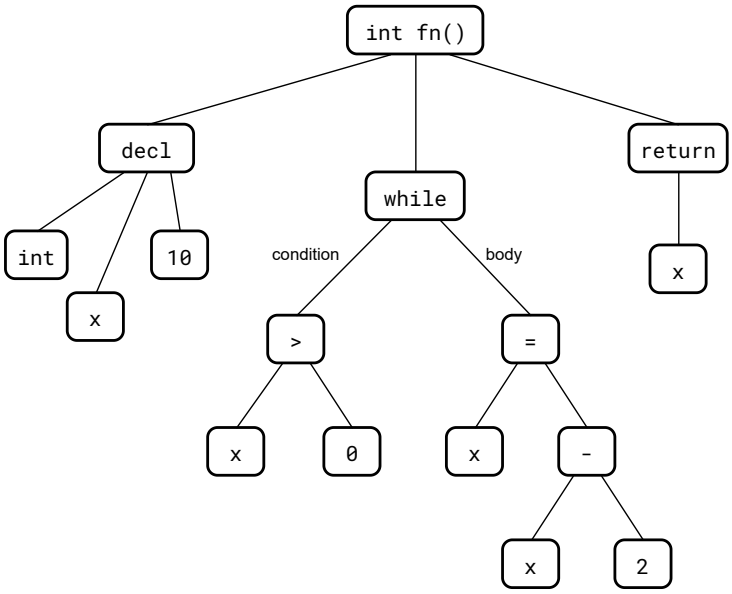

Figure 3.1: AST of function `fn`.

CFGs (Figure 3.2) are directed graphs that describe the flow of a program, where each node represents a sequence of statements (usually a basic block) and each edge depicts a possible control flow. This representation provides a detailed view of the program, emphasizing the program's control flow.

Notice how the AST highlights the hierarchical organization of the code while the CFG focuses on the flow of the program, particularly in the `while` statement.

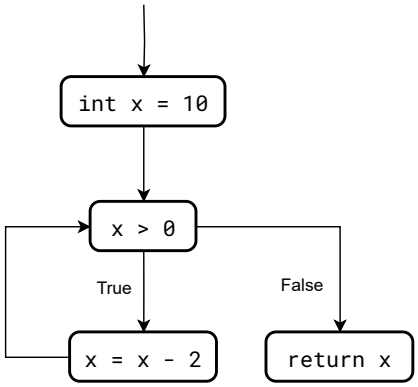

Figure 3.2: CFG of function `fn`.

3.2 A simple language

Let us suppose we have a very simple language. This language includes function definitions and six fundamental operations: the operators add ($+$) and less-than ($<$), the declaration (`let`), the `if` statement, the `assert` and function calls. We can use these operations to create very simple programs, such as the one below.

```
1  let v1 = 1 in
2      let v2 = v1 + 1 in
3          assert(v2 > v1)
```

In this program, we initialize the variable `v1` with `1`, and then we define the variable `v2` with the value `v1 + 1`, which evaluates to `2`. Subsequently, on the third line, we perform an assertion to check if `v2` is less than `v1`. At run time, this always evaluates to `true` and therefore the assertion is always valid.

We can define a static analysis for type-checking programs written in this language. A program is considered well-typed if all assertions in the program are valid. For instance, in the program above we have that `v2 = v1 + 1 = 1 + 1 = 2 > v1` and therefore the `assert` is always valid and the program is considered well-typed. We define the lattice and abstract domain of the analysis in Section 3.3 and the typing rules in Section 3.4, as we introduce the concepts.

3.3 Lattices and abstraction domains

A partial order is a pair $(S, \sqsubseteq)$ in which S is a set and $\sqsubseteq$ is a binary relation, where $x \sqsubseteq y$ represents that x is *at least as precise* as y.

Given $s, x, y \in S$, if s is an upper bound of x and y ($s \sqsubseteq x \sqcup y$) then it means that s is at least as precise as both x and y. The notation $x \sqcup y$ is called the *join* of x and y and it is equivalent to the *least upper bound* of x and y. Similarly, we can declare that s is a lower bound of x and y ($x \sqcap y \sqsubseteq s$), where

$x \sqcap y$ is called the *meet* of x and y and is the same as the *greatest lower bound* of x and y.

When designing a static analysis tool, it is important to select an appropriate abstract domain. This is a set of values used to represent the possible states of a program variable at a given point in the program execution. There are different types of abstract domains, with different characteristics. The choice of the abstract domain depends on the analysis and the program being analyzed.

The abstract domain can be represented in an abstract structure called a lattice. A lattice is a partial order $(S, \sqsubseteq)$ in which $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in S$. The *least upper bound* is the operation used when combining abstract information from multiple sources, for instance when merging `if` branches.

A static analysis may have to handle uncertain information. In a lattice, an unknown value is represented by $\top$ (called "top"), which is the topmost element in the lattice. Equivalently, the analysis may also come across expressions whose values are unreachable, which are represented by $\bot$ (called "bottom"), in the bottom of the lattice.

For the analysis of the language defined in Section 3.2, the chosen abstract domain is integer intervals. In this domain, each abstract value is of the form [x, y]. The analysis should infer these intervals for each variable so that, if the abstract value for variable v is [x, y], then it is true that $\mathtt{x} \leq \mathtt{v} \leq \mathtt{y}$. The $\bot$ represents an empty interval, whereas the $\top$ represents the interval $(-\infty, +\infty)$. The corresponding lattice is displayed in Figure 3.3.

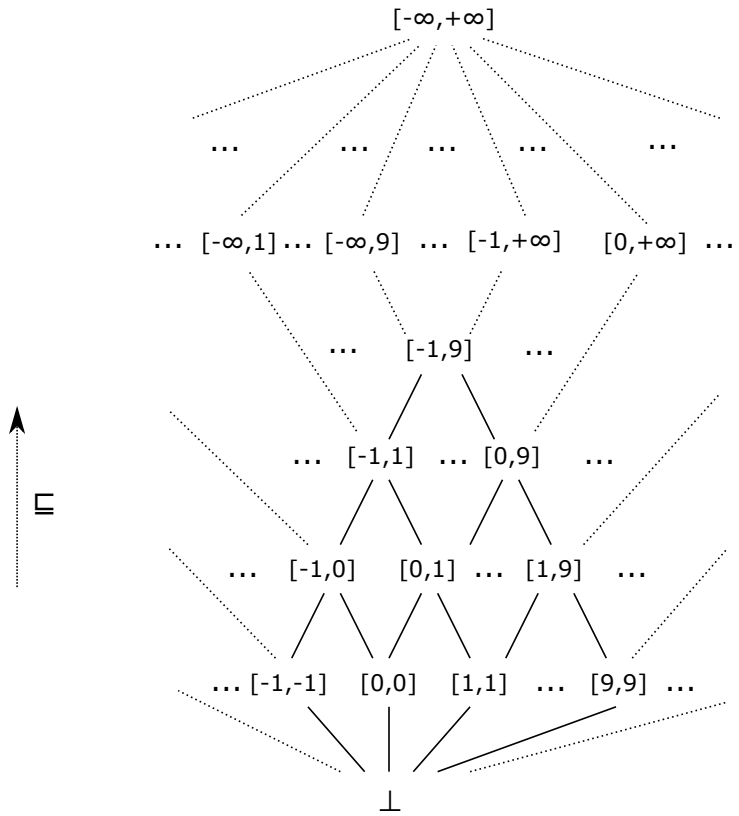

Figure 3.3: Hasse diagram for interval lattice [1].

The binary relation $\sqsubseteq$ in the lattice indicates that the combination of two abstract values yields a more general value, and therefore, potentially leading to a loss of precision. The following example illustrates this loss of precision:

```
1  λ x .
2      let v1 = if x > 0 then 3 else 1 in
3          let v2 = if x > 0 then 2 else 0 in
4              assert(v1 > v2)
```

In the code snippet, when `x > 0`, `v1 = 3` and `v2 = 1`, making the `assert` on the fourth line valid. Equivalently, when `x <= 0`, `v1 = 2` and `v2 = 0`, also making the `assert` correct. Therefore, the `assert` will never fail at run time.

However, the situation changes when conducting static analysis, with the chosen abstract domain. Assuming that we have a single state for the entire function, we want to find all possible values for each variable and store them in an interval. These inferred intervals are then used on the fourth line to validate the `assert`.

In the code snippet in question, on the second line, we have that the value of `v1` may be either `3` or `1`. The analysis merges these two possible values into a single interval, resulting in the interval [1, 3]. Similarly, on the third line, the value of `v2` may be either `2` or `0` and thus the analysis infers the interval [0, 2]. Then, these intervals are used to validate the `assert`. In this case, the analysis determines whether [1, 3] $>$ [0, 2]. Since the intervals intercept, the specification of the analysis defines whether it should accept or reject the `assert`. It is important to note, however, that the analysis cannot guarantee the correctness of the `assert`, due to loss of precision.

3.4 Typing rules

Previously, we specified the abstract domain for the language defined in Section 3.2. Now, we can define the set of typing rules that specify the static analysis for type-checking programs written in this language. These are inference rules that describe the analysis and can be applied in programs to determine if they are well-typed.

$$\frac{c \text{ is constant}}{\Gamma \vdash c \in [c,c]}\ (\textsc{Const}) \qquad \frac{\Gamma : x \in [a,b]}{\Gamma \vdash x \in [a,b]}\ (\textsc{Var}) \qquad \frac{x \notin \Gamma}{\Gamma \vdash x \in (-\infty,+\infty)}\ (\textsc{Sym})$$

$$\frac{\Gamma, x : \tau_1 \vdash p : \tau_2}{\Gamma \vdash (\lambda x.\ p) : \tau_1 \to \tau_2}\ (\textsc{FuncDef}) \qquad \frac{\Gamma \vdash p_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash p_2 : \tau_1}{\Gamma \vdash p_1\ p_2 : \tau_2}\ (\textsc{FuncCall})$$

$$\frac{\Gamma \vdash x : \tau_1 \qquad \Gamma, y : \tau_1 \vdash p : \tau_2}{\Gamma \vdash (\texttt{let}\ y = x\ \texttt{in}\ p) : \tau_2}\ (\textsc{Let}) \qquad \frac{\Gamma \vdash x \in [a,b] \qquad \Gamma \vdash y \in [c,d]}{\Gamma \vdash x + y \in [a+c, b+d]}\ (+)$$

$$\frac{\Gamma \vdash x \in [a,b] \qquad \Gamma \vdash y \in [c,d] \qquad b < c}{\Gamma \vdash x < y \in \{True\}} \ (<_T) \qquad \frac{\Gamma \vdash x \in [a,b] \qquad \Gamma \vdash y \in [c,d] \qquad d \leq a}{\Gamma \vdash x < y \in \{False\}} \ (<_F)$$

$$\frac{\Gamma \vdash x \in [a,b] \qquad \Gamma \vdash y \in [c,d] \qquad [a,b] \cap [c,d] \neq \emptyset \qquad a \neq d}{\Gamma \vdash x < y \in \{True, False\}} \ (<_?)$$

$$\frac{\Gamma \vdash c \in \{True\} \qquad \Gamma \vdash x : \tau}{\Gamma \vdash (\texttt{if}\ c\ \texttt{then}\ x\ \texttt{else}\ y) : \tau} \ (\textsc{IfThen}) \qquad \frac{\Gamma \vdash c \in \{False\} \qquad \Gamma \vdash y : \tau}{\Gamma \vdash (\texttt{if}\ c\ \texttt{then}\ x\ \texttt{else}\ y) : \tau} \ (\textsc{IfElse})$$

$$\frac{\Gamma \vdash c \in \{True, False\} \qquad \Gamma \vdash x : \tau_1 \qquad \Gamma \vdash y : \tau_2}{\Gamma \vdash (\texttt{if}\ c\ \texttt{then}\ x\ \texttt{else}\ y) \in \tau_1 \sqcup \tau_2} \ (\textsc{If}_?) \qquad \frac{\Gamma \vdash c \in \{True\}}{\Gamma \vdash \texttt{assert}(c) \in \{True\}} \ (\textsc{Assert})$$

We can now use these rules to validate programs written in this language. To formally verify the correctness of a program, we can recursively apply these typing rules, forming a tree structure known as a derivation tree. If the tree is successfully constructed, it means that the program is well-typed. For instance, consider the following program:

```
1  let v1 = 1 in
2      let v2 = if v1 < 0 then 2 else 0 in
3          assert(v2 < v1)
```

Below is a complete derivation tree, therefore serving as a formal proof that the assert statement will always succeed.

$$\dfrac{\dfrac{1 \text{ is constant}}{\emptyset \vdash 1 \in [1,1]}\ (\textsc{Const}) \qquad \dfrac{\texttt{(1)} \quad \texttt{(2)}}{v1 \in [1,1] \vdash \texttt{let}\ v2 = \texttt{if}\ v1 < 0\ \texttt{then}\ 2\ \texttt{else}\ 0\ \texttt{in}\ \texttt{assert}(v2 < v1) \in \{True\}}\ (\textsc{Let})}{\emptyset \vdash \texttt{let}\ v1 = 1\ \texttt{in}\ \texttt{let}\ v2 = \texttt{if}\ v1 < 0\ \texttt{then}\ 2\ \texttt{else}\ 0\ \texttt{in}\ \texttt{assert}(v2 < v1) \in \{True\}}\ (\textsc{Let})$$

Proof of the branch `(1)`:

$$\dfrac{\dfrac{\dfrac{v1 \in [1,1] : v1 \in [1,1]}{v1 \in [1,1] \vdash v1 \in [1,1]}\ (\textsc{Var}) \qquad \dfrac{0 \text{ is constant}}{v1 \in [1,1] \vdash 0 \in [0,0]}\ (\textsc{Const}) \qquad 0 \leq 1}{v1 \in [1,1] \vdash v1 < 0 \in \{False\}}\ (<_F) \qquad \dfrac{0 \text{ is constant}}{v1 \in [1,1] \vdash 0 \in [0,0]}\ (\textsc{Const})}{v1 \in [1,1] \vdash (\texttt{if}\ v1 < 0\ \texttt{then}\ 2\ \texttt{else}\ 0) \in [0,0]}\ (\textsc{IfElse})$$

Proof of the branch `(2)`:

$$\dfrac{\dfrac{\dfrac{v1 \in [1,1],\ v2 \in [0,0] : v2 \in [0,0]}{v1 \in [1,1],\ v2 \in [0,0] \vdash v2 \in [0,0]}\ (\textsc{Var}) \qquad \dfrac{v1 \in [1,1],\ v2 \in [0,0] : v1 \in [1,1]}{v1 \in [1,1],\ v2 \in [0,0] \vdash v1 \in [1,1]}\ (\textsc{Var}) \qquad 0 < 1}{v1 \in [1,1],\ v2 \in [0,0] \vdash v2 < v1 \in \{True\}}\ (<_T)}{v1 \in [1,1],\ v2 \in [0,0] \vdash \texttt{assert}(v2 < v1) \in \{True\}}\ (\textsc{Assert})$$

It is important to note that derivation trees are effective in proving the validity of small and simple

programs. However, when dealing with complex or non-terminating programs, derivation trees become impractically large or even infinite. In those cases, an alternative is to validate these programs by employing a least fixed-point algorithm.

3.5 Flow-sensitivity

Static analysis tools can adopt either of two approaches regarding flow-sensitivity: flow-insensitive or flow-sensitive analysis. Flow-insensitive analysis ignores the control flow within a program. It computes a single state for each function or block and, consequently, disregards the order of statements inside that function or block. On the other hand, flow-sensitive analysis, such as dataflow analysis, takes into account the control flow and the statements order of the program's source code. It computes a different state for each program point.

In general, flow-sensitive analyses are more precise but also more computationally expensive and less scalable than flow-insensitive analyses. Flow-sensitive analyses require storing states at several different program points and generating a different state for each branch in the execution, which significantly affects memory usage. On the other hand, flow-insensitive analyses sacrifice precision by having a single global state per function or block, but spare time and memory overhead.

Below is a very simple function written in C. It declares a variable `x` with the value `-1` and afterward assigns to it the value `1`. Finally, the `assert` instruction verifies whether variable `x` is larger than 0.

```
1  void fn() {
2      int x = -1;
3      x = 1;
4      assert(x > 0);
5  }
```

The `assert` in the third line never fails at run time, since, at this point in the code, the value of `x` is always `1`. However, when using static analysis to verify the program, it may be classified as invalid, depending on whether the analysis is flow-insensitive or flow-sensitive. Suppose the chosen abstract domain is integer intervals, as in Section 3.3. If the analysis is flow-sensitive, it computes a different state for each program point. Therefore, when the analysis reaches the `assert` statement, it accurately assumes that the value of `x` is `1`.

On the other hand, if the analysis is flow-insensitive, then a single state is computed for the entire function. Consequently, the analysis merges all possible values of `x`, in this case, inferring that `x` belongs to the interval [-1, 1]. Therefore, a flow-insensitive analysis cannot guarantee the correctness of the program in question, being less precise when compared to a flow-sensitive alternative.

3.6 Context-sensitivity

A static analysis is intra-procedural if it analyzes the body of a single function at a time, isolated from the rest of the program. This is also referred to as context-insensitive analysis [13]. On the contrary, a static analysis is inter-procedural or context-sensitive if it considers the entire program's structure and involves inspecting the context and dependencies between procedures.

Inter-procedural analyses may be more precise but are slower than intra-procedural analyses.

3.7 Field-sensitivity

Field-sensitivity [14] determines how structs are handled by static analysis, presenting two possible approaches. The first one is field-insensitive and dictates that different fields in the same struct should all be represented by the same abstract value. Alternatively, a static analysis can be field-sensitive, employing a more complex lattice where each field has its own corresponding independent value.

Similar to flow-sensitivity and context-sensitivity, a field-sensitive analysis is more precise but comes at the cost of higher memory and time overhead.

In the following program, we define a struct `S` which holds two integers. In the function `main`, we declare `s` and initialize it with the integers `-1` and `1`.

```
1  struct S {
2      int x;
3      int y;
4  };
5
6  int main() {
7      struct S s = {-1, 1};
8      assert(s.y > 0);
9      return 0;
10 }
```

The `assert` inside the function verifies whether the field `y` of `s` is larger than `0`. This `assert` always passes at run time since the value of the field `y` is `1`.

When we submit this program to static analysis, using integer intervals as the abstract domain, only a field-sensitive approach will be able to accurately verify it. If the analysis is field-insensitive instead, it will hold a single interval for the whole struct. In this context, the inferred interval would be [-1, 1], which is not precise enough to validate the `assert`.

3.8 False positives vs false negatives

A static analysis tool may admit false negatives and false positives because it typically needs to be scalable, precise, and fast, requiring approximation. A false positive happens when the analysis incorrectly reports a bug that does not exist. A static analysis tool that does not admit false positives is complete. On the other hand, a static analysis tool is sound if it admits no false negatives. These happen when a real bug is not reported by the analysis. A sound analysis will never miss errors, but may reject a valid program. If an analysis is complete but not sound, then it guarantees that, if it reports an error, then that error is not a false positive. However, it may miss some errors.

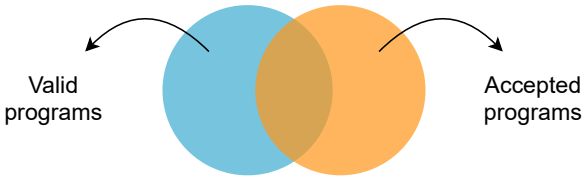

Figure 3.4: False positives and false negatives.

Figure 3.4 helps visualize these definitions. The valid programs are represented by the blue circle while the accepted programs are illustrated by the orange one. The valid programs that are correctly accepted by the analysis correspond to the intersection of the two circles. The false positives correspond to the blue part that does not coincide with the orange circle ($V \backslash A$), whereas the orange circle minus the blue one ($A \backslash V$) represents the false negatives.

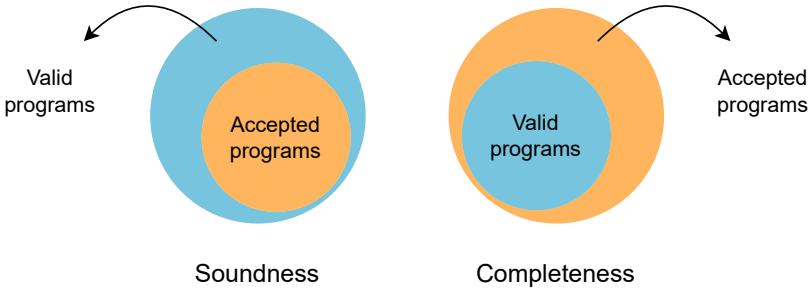

Figure 3.5: Soundness (left) and Completeness (right).

Figure 3.5 uses the same diagrams to help visualize the concept of soundness (absence of false negatives) and completeness (no false positives). In the soundness diagram, there are no false negatives and the false positives correspond to the blue region that does not intercept the orange region. Similarly, in the completeness diagram, there are no false positives and the false negatives are represented by the orange region except the blue one.

As discussed previously, static analysis tools have to make assumptions as they evaluate the code. These assumptions make the tool lose precision and therefore are responsible for generating false

positives and false negatives. If a static analysis tool produces false negatives, the absence of errors reported by the tool is not enough to guarantee that the code is valid. Equivalently, if a tool reports too many false positives, it becomes impractical to be used in production environments. In fact, according to Microsoft, a high number of false positives is a major barrier to the adoption of static analysis tools [15, 16]. Consequently, the number of either false positives or false negatives created by a static analysis tool should be reasonable.

The ASSERT rule, as described in the typing rules above (Section 3.2), determines that the analysis allows for false positives. This means that when the condition c can be either $True$ or $False$ ($c \in \{True, False\}$), the assert fails even though the program can be valid. This rule may have a well-typed program incorrectly classified as ill-typed.

For instance, the analysis of the following code snippet may generate a false positive.

```
1  λ x .
2      let v1 = x > 0 then 1 else -1 in
3          assert(v1 < 0)
```

The condition `x > 0` may be either true or false, which means that `v1` $\in [-1, 1]$. The `assert` fails because `v1` may be larger than 0 and thus the analysis rejects this program. However, if `x` is less than 0, then the program is valid and the analysis would generate a false positive.

We could change the existing ASSERT rule to prevent the `assert` statement from failing when we cannot establish that c is always $False$. The ASSERT rule would then be the following:

$$\frac{\exists\, S : (\Gamma : c \in S \wedge True \in S)}{\Gamma \vdash \texttt{assert}(c) \in \{True\}} \quad (\textsc{Assert}_{\textsc{New}})$$

This change would make the analysis admit false negatives instead of false positives, ensuring that a program cannot be flagged as ill-typed when it is valid. However, this means that the analysis may incorrectly label invalid programs as well-typed. For instance, the analysis would now accept the previous code snippet because there is a scenario where the condition `v1 < 0` is true. However, this may be a false negative if `x` is greater than 0, which makes the program invalid.

Typically, it is not possible to define a static analysis that is simultaneously sound and complete.

3.9 Challenges of static program analysis

One of the main challenges of static program analysis is efficiency [17]. A static analysis tool should be capable of providing valuable results within a reasonable amount of time. If the analysis of a program takes too long, the tool becomes impractical.

Aside from being efficient, a static analysis tool should also be scalable. It should be able to handle large and complex programs, while still providing results within an acceptable time.

Another challenge when developing static program analysis is achieving high precision. Ideally, the analysis should be sound and complete. A high number of false negatives and false positives makes the tool unreliable and thus may discourage developers from adopting it. High precision is hard to achieve because of the size and complexity of some programs. Analyzing all the possible executions of such programs results in high memory and run-time overhead. Nonetheless, unsound tools are still useful in identifying errors in programs, even though they might also accept invalid programs. Similarly, while incomplete tools may reject some valid programs, they can still provide guarantees about the correctness of some programs. Given that it is unrealistic to have a static analysis that is both sound and complete, the majority of static analysis tools aim for soundness, while trying to minimize the number of false positives.[1]

There is a trade-off between precision and efficiency in static program analysis and finding the right balance between them is very challenging. An analysis may be fast, but then it is unlikely to be sufficiently precise. Likewise, a very precise analysis probably will not terminate in a reasonable time for large and complex programs.

[1] http://soundiness.org/

4

Related Work

Contents

In this chapter, we examine various techniques for improving the safety of programs, with a focus on preventing memory bugs and vulnerabilities.

There are multiple memory error detection tools, each with its unique features and capabilities. These tools differ in many aspects, such as performance, the type of memory bugs they can detect, and the amount of memory they require. It is worth noting that there is often a trade-off between the overhead of a tool (in terms of performance and memory usage) and the type of bugs it can detect.

Below we describe several memory error detection tools, highlighting their key features and downsides.

4.1 Static analysis

Static analysis tools analyze the code of a program to identify issues without executing the code. These tools can find a wide range of errors, including memory safety issues such as dangling pointers, uninitialized variables, and buffer overflows.

Coverity [16], Astrée [18], and Frama-C [19] are static analysis tools that find bugs in the source code of large software systems, including errors related to memory safety. These tools are designed to help developers improve the safety and quality of their code written in C or other languages.

Coverity is designed to be used during the development process to identify potential bugs in the code. It performs static analysis without making any changes to the code and provides full path coverage, guaranteeing that most lines of code and most possible execution paths are tested [20]. This tool is popular among developers because it is fast, accurate, and reliable, with a low number of false positives. However, Coverity is not designed to guarantee the absence of errors, but rather to find as many as possible, and thus it is not sound. This tool also allows developers to create their own checkers, but the APIs are complex, and hence the process can be difficult and prone to errors. Additionally, Coverity includes lightweight versions that can be integrated with IDEs, allowing it to analyze and identify issues in the code while it is being written.

Astrée is specifically designed to be used in safety-critical software systems, such as those used by Airbus. This tool uses a combination of data-flow analysis and abstract interpretation techniques. Data-flow analysis is a technique that is used to analyze how data is transformed and used throughout the program. Astrée operates on multiple abstract domains, such as interval, octagon, and memory abstract domains, allowing it to identify a wide range of errors.

Finally, Frama-C uses flow-sensitive abstract interpretation to compute a set of possible values for each variable at every point in the program. For example, it can be used to verify that no input can make the program generate a run time error. This tool requires developers to provide extra information, namely invariant, precondition, and postcondition annotations. These annotations are used to specify properties and conditions that are expected to hold for variables and functions during the execution of the program. These are checked during the analysis of the code and, if any of these annotations are violated, the tool reports it. Frama-C allows developers to explore the program's structure and compute metrics on it.

At a different level, Microsoft created a source-code annotation language (SAL)[1] that allows developers to add annotations to C and C++ code. These annotations can be used to describe a set of properties and requirements that cannot be expressed through C/C++ code alone. For instance, annotating function parameters with the annotations `_In_` and `_Out_` indicates how they should be handled by the function. A parameter annotated with `_In_` is an input parameter and therefore should be initialized

[1] https://learn.microsoft.com/en-us/cpp/code-quality/annotating-function-parameters-and-return-values

before a function call, whereas a parameter annotated with `_Out_` is an output parameter which means that it should be writable. Another example is shown below:

```
1  void bar(_Out_writes_(p_size) char* p, _In_ size_t p_size) {
2      // ...
3  }
```

The function `bar` receives a pointer `p` to a char array and a size `p_size`. The annotation `_In_` indicates that `p_size` is an input parameter of function `bar` and the annotation `_Out_writes_(p_size)` specifies that the char array pointed by `p` has, at least, `p_size` elements and that `p` will be written by the function.

The compiler uses these annotations to check if the code follows the specified properties, reporting any violations to them. This allows the analysis to be intra-procedural instead of inter-procedural, as is the case with lifetime annotations in Rust, increasing the efficiency of the analysis. The use of SAL also allows automated static analysis tools to be more accurate when analyzing C/C++ code, resulting in fewer false positives and false negatives. For this reason, these annotations help reduce memory errors in C/C++ code.

The C++ programming language includes a set of concepts that provide safer handling of sequences of objects, specifically views, which include `std::span` and `std::string_view`, and checked arrays, namely `std::array`. These classes can help prevent some spatial memory errors.

Clang has a set of annotations similar to those in SAL, but specific for the analysis of concurrent programs [21]. For instance, a variable called `balance` can be annotated with `GUARDED_BY` to indicate that it should only be read or written to after acquiring a mutex `mu`.[2]

```
1  int balance GUARDED_BY(mu);
2
3  void changeBalance(int amount) {
4      balance += amount;   // warning
5  }
```

In line 4, the annotation `GUARDED_BY` is violated, and thus Clang would report this issue.

In conclusion, static analysis tools are useful for improving code safety and quality. However, as discussed in Section 3.9, they come with three significant challenges, namely efficiency, scalability, and accuracy.

4.2 Memory debugging tools

Memory debugging tools help identify and fix issues related to memory usage in a program by executing the code. They can be used to detect uninitialized memory access and buffer overflows, for instance.

[2]This example was taken from https://clang.llvm.org/docs/ThreadSafetyAnalysis.html

Many of these tools use shadow memory in order to store metadata information about memory allocations, such as which bits are initialized or the bounds of each pointer. This information is used on each load/store operation, during execution, to detect memory bugs.

Some tools, like AddressSanitizer [22] and MemorySanitizer [23], implement compile-time instrumentation. Thus, they are reasonably fast but do not handle external libraries and require the program to be recompiled whenever the tool is used. On the other hand, Valgrind [24] performs binary instrumentation at the machine code level, during program execution. This approach is much slower but works with any external libraries and does not require recompilation.

Valgrind detects both spatial and temporal errors, with typically no false positives. Due to its complexity, it has an average slowdown of 22x. AddressSanitizer and MemorySanitizer each focus on only one kind of bug, instead. MemorySanitizer detects only uses of uninitialized memory and only rarely results in false negatives. AddressSanitizer, on the other hand, is designed to find spatial errors. It does so by surrounding all objects with redzones and by implementing quarantine. Redzones are areas of memory around objects that are made unaddressable or "poisoned", helping to detect memory accesses outside of the bounds of the respective object. Quarantine, on the other hand, involves delaying the reallocation of recently freed memory, by temporarily marking it as "poisonous". This helps to prevent use-after-free bugs. However, the detection of spatial errors in AddressSanitizer is probabilistic and hence misses some errors. For instance, it misses bugs when the out-of-bounds access is outside the respective redzone and when the quarantine period is not long enough, due to large amounts of memory being allocated and deallocated. AddressSanitizer also has a performance overhead of almost 2x. Overall, Valgrind is much slower than its alternatives, being slower than running AddressSanitizer plus MemorySanitizer.

Regarding memory overhead, all of these tools greatly increase the amount of memory used. Valgrind and MemorySanitizer both need a large amount of shadow memory to detect uses of uninitialized memory, storing one bit of shadow memory for each bit of application memory. AddressSanitizer only uses 1/8 of the program's memory for the shadow memory, but even so, it has a memory overhead of almost four times. This memory overhead is primarily due to the implementation of redzones and quarantine.

4.3 Code instrumentation tools

Code instrumentation tools add additional code to a program with the purpose of monitoring or improving its behavior. Specifically, the code added can be used to track memory usage and check for memory errors, thus helping mitigate certain types of memory safety problems.

While debugging tools are typically used after a problem has been identified, code instrumentation

tools, on the other hand, can be used to prevent problems from occurring in the first place.

There are two kinds of code instrumentation tools: object-based and pointer-based. Both approaches detect spatial errors but they complement each other in advantages and disadvantages.

Object-based approaches, such as SAFECode [25] use a separate lookup structure for storing each object's metadata. Therefore, they do not change the memory layout or pointer representation but fail to find some spatial errors (namely sub-object overflows). SAFECode's run time overhead can be as high as 30%.

On the other hand, pointer-based approaches replace all pointers with corresponding fat pointers to store some properties, such as the size and the type of the values of the pointer. The use of fat pointers has the consequence of changing the memory layout. However, this implementation allows having multiple pointers to the same object, where each one has a different base and/or bound. For this reason, pointer-based approaches detect sub-object overflows.

CCured [26] is a pointer-based approach that uses type inference to statically determine which pointers require bound checking. Then, it performs dynamic checks only on the unsafe pointers, consequently reducing the performance impact. Even so, its overhead can be as high as 150%. CCured ensures complete protection against spatial errors but requires manual modifications to the code.

Softbound [27] is another pointer-based approach that implements a separate metadata lookup structure, like object-based approaches. Therefore, Softbound provides complete spatial safety without changing the memory layout. Yet, it has an overhead of 67% on average.

Finally, Checked C [28] is also a pointer-based approach that incrementally adds spatial safety to programs (contrary to all other tools seen up until now). This is achieved by allowing the programmer to create checked regions, where the tool applies the dynamic checks, and unchecked regions, which are expected to be safe. Checked C requires no changes to the program's pointer representation and has a relatively small overhead of 8.6%. However, on average, 17.5% of lines of code have to be manually modified, which is not realistic for applications with millions of lines.

4.4 Software-based production tools

Most software tools discussed previously sustain big slowdowns and hence are not suitable for use in production environments. However, they can be adapted to be used in production. For instance, Google uses AddressSanitizer [29] with sampling to find bugs in Chrome and Android in production, decreasing the tool's overhead. The idea is that only 0.001% of memory allocations are monitored by AddressSanitizer, reducing both the overhead and the probability of finding bugs. However, there are billions of devices using Chrome and Android, thus the likelihood of finding these bugs is still very high.

Alternatively, there is a set of software tools specifically designed to be used in production, such as

DangZero and Control-Flow Integrity (CFI) tools.

DangZero [30] is a tool that detects use-after-free vulnerabilities. The main idea is to give the program direct access to the page table and use each page table entry's metadata to record whether the corresponding pointer is free or not. It has a relatively small performance slowdown of 16% on average. However, it presents other limitations regarding compatibility. It is only compatible with some Linux kernel versions and uncommon pointer encodings and memory allocators are not supported.

Control-flow integrity is a technique that aims to prevent attackers from manipulating the control flow of a program during execution [31]. This is achieved by enforcing a predetermined control-flow graph that defines all the possible valid paths that the program can take. To enforce Control-Flow Integrity, run-time checks are added to the binary. These dynamically validate the run time execution, blocking the attacker from executing any instructions outside the legal control-flow graph.

This technique is based on the assumption that only certain control flows should be expected during the execution of a program. This technique can provide strong guarantees against attackers and it is simple and verifiable. However, it also has an impact on the binary size.

Control-flow integrity is already implemented in the GCC, MSVC, and LLVM compilers [32]. This implementation does not generate false positives and has a run time overhead lower than 10% in most cases and thus may be suitable for deployment in production.

Microsoft has developed a variant of the previous technique, called data-flow integrity [33]. This approach uses static analysis to compute a data-flow graph, which represents how data is transformed and used throughout the program. Then, during run time, it ensures that the program does not execute data flows that fall outside of the data-flow graph previously determined. While control-flow integrity only targets control-data attacks, data-flow integrity aims to protect against both control-data attacks and non-control-data attacks. The latter is a type of software vulnerability that allows attackers to modify data values, such as corrupting data structures and injecting malicious code.

Control-flow integrity has multiple benefits, such as low overhead and no false positives. For this reason, it is used in production, being implemented both in Chrome and Windows. However, it also has some weaknesses. For instance, control-flow integrity usually does not prevent all types of attacks and errors, since the assumptions about the program's execution flow may not hold in all cases. Additionally, in some cases, control-flow integrity may require instrumentation to create the control-flow checks. This instrumentation requires the modification of the program's source code, making the program more complex and difficult to maintain. However, in the general case, source code modification is not required.

4.5 Hardware-based solutions

Hardware-based production tools use the hardware features of a system to help prevent memory errors and improve the memory safety of software. These tools can be used in production environments, which requires them to be reliable, stable, and fast.

CHERI [34] is a system that provides spatial memory safety to C and C++ by using fat pointers, called capabilities. CHERI consists of special registers, which hold the capabilities, and a set of capability-aware instructions that extend the processor's original instruction-set architecture. These special instructions load, store, and manipulate CHERI's capabilities.

The main issue with CHERI is its lack of compatibility. The use of a new set of instructions requires either increasing the opcode space or creating a new opcode encoding. Thus, CHERI is architecture-dependent and only incrementally adoptable. Moreover, it might require some code changes to C and C++ programs and further adaptation is necessary if the program uses a specific memory allocator, not supported by CHERI.

In a similar fashion to CHERI, Intel MPX [35] also comprises a set of new instructions and registers that aim to provide spatial memory safety. However, Intel MPX presents flaws concerning both performance and usability. It is slower than many software-based tools, with an average slowdown of 50%, and has a memory overhead of 2x, on average. It has false positives and it has conflicts with other instruction-set architecture extensions. Finally, Intel MPX does not support many C/C++ libraries and thus, around 80% of programs do not build or execute correctly. In fact, Intel MPX has been officially discontinued.[3]

Intel SGX [36] is a set of guard extensions to the Intel architecture that attempt to provide integrity and confidentiality guarantees when running code on a remote machine that is not trusted. To achieve this, Intel SGX uses trusted hardware in the remote computer, which creates a secure container or enclave to protect and isolate the data used in the computation. However, its guarantees are not strong enough, given that it does not protect against multiple sophisticated attacks such as software side-channel attacks, cache timing attacks, and passive address translation attacks.

4.6 Summary

In this section, we explored different techniques for detecting and preventing memory safety bugs and vulnerabilities. However, none of these approaches is perfect. Most of these tools focus on detecting spatial errors, while much fewer tools address temporal errors. Static analysis tools, in general, have theoretical limitations. In addition, some tools, such as Valgrind and MemorySanitizer, are not suitable

[3] https://tinyurl.com/2b2e59fb

for production environments due to their performance and memory overheads. Others produce too many false positives or false negatives, respectively crashing unnecessarily or missing important errors.

As previously mentioned, memory safety errors can be classified into spatial errors and temporal errors. Preventing these errors requires different approaches and hence many tools are designed to address only one type of these errors.

Spatial safety requires maintaining information about the bounds of each object. This information can then be deleted when the object is freed. Temporal safety, on the other hand, involves keeping information related to the lifetime of each object throughout the entire program, since the program may try to access that object at any time, even after the object is no longer valid. If the metadata for every alive and dead object were kept throughout the entire program, it would result in significant memory usage. Thus, this metadata eventually has to be cleaned up to avoid excessive memory overhead, meaning that temporal safety tools have to be probabilistic. The decision of when to delete the metadata of an object depends on the tool, affecting the tool's likelihood of finding temporal memory bugs, but also its performance and memory overhead.

Therefore, temporal safety is generally much harder to ensure than spatial safety. As a result, temporal safety tools usually have a higher overhead and are more challenging to implement, which explains why there are fewer tools available to address temporal memory errors.

In Rust, spatial safety is guaranteed by using bound checks at run time and by leveraging the type system to ensure that out-of-bounds bugs cannot be expressed in source code. Temporal safety, on the other hand, is enforced at compile time by the use of lifetimes, which eliminate errors such as dangling references. Since Rust is a memory-safe language and requires lifetimes to verify that safety, the code will not compile if the lifetimes are incorrect or unknown. Therefore, in order to achieve interoperability between C++ and Rust, it is necessary to add the concept of lifetime annotations to C++ code.

This thesis aims to address temporal memory errors, which is the most challenging type of memory safety. It focuses on the concept of lifetime annotations and their implementation in C++, contributing to the interoperability between C++ and Rust.

5

Lifetime Analysis in C++

Contents

In this chapter, we present the design of Lifetimes Static Analyzer (LSA), whose goal is to verify Rust-like lifetime annotations in C/C++, ensuring they are valid according to Rust's lifetime rules. To achieve this, it is crucial for LSA to closely emulate Rust's lifetime analysis and produce the same results. For this reason, before we delve into our C/C++ lifetime analysis, it is important to understand Rust's lifetime analysis.

In the previous chapters, we primarily focused on C++ when discussing memory-safety vulnerabilities and interoperability with Rust. However, given that C++ is essentially a superset of C, the majority of what was covered applies to C as well, including the Rust-like lifetime annotations. Nevertheless, C++ is significantly more complex than C. It includes intricate constructs such as classes, templates, and lambda functions, requiring a more extensive and intricate implementation of LSA. Due to time constraints, we opted to restrict our analysis to C constructs. It is worth noting that LSA supports the entire scope of the C programming language, except for the heap, as discussed in Section 8.6. Since

C++ is an extension of C, LSA is also capable of analyzing simple C++ programs.

There are multiple language differences between C/C++ and Rust, such as pointer aliasing. In C/C++ there can be two or more non-constant references pointing to the same value, whereas in Rust there can be either one mutable reference or multiple immutable references. Consequently, contrarily to Rust, a C/C++ lifetime analysis must handle pointer aliasing. In these cases, we cannot stick to Rust's lifetime analysis design and instead have to come up with our own solution, which should still be compatible with Rust's analysis.

This chapter starts with an overview of Rust's lifetime analysis, followed by the specification of our C/C++ lifetime analysis. We start by introducing the foundation of the analysis, which is simpler, in Section 5.2, and then we explore the more complex parts, in Section 5.3. LSA's implementation is explained in detail in Chapter 6.

5.1 Rust's lifetime analysis

In order to understand Rust's lifetime analysis, we will gradually explore more complex Rust code snippets and examine the warnings generated in each instance.

Below is the very simple function `f1`, which takes a reference `x` and returns that same reference.

```
1 fn f1<'a, 'b>(x: &'a i32) -> &'b i32 {
2     return x;
3 }
```

The reference `x` is annotated with lifetime `'a` and the function is supposed to return a value with lifetime `'b`. When we try to compile this code, Rust returns the following error:

```
error: lifetime may not live long enough
 --> src/lib.rs:2:12
  |
1 | fn f1<'a, 'b>(x: &'a i32) -> &'b i32 {
  |       --  -- lifetime `'b` defined here
  |       |
  |       lifetime `'a` defined here
2 |     return x;    // error
  |            ^ function was supposed to return data with lifetime `'b`
  |              but it is returning data with lifetime `'a`
  |
  = help: consider adding the following bound: `'a: 'b`
```

The fundamental rule for Rust's lifetime analysis is that a reference cannot live longer than the value that it points to. Consequently, the value returned by the function `f1` has to live longer than lifetime `'b`. If the compiler could prove that the lifetime of `x` (`'a`) outlived lifetime `'b`, the code would be accepted (note that this is what the error message suggests in order to fix the error). However, without such information, the Rust compiler cannot prove that the code is correct and, for that reason, generates the above error.

All references in function parameters must be annotated with a lifetime, which may be elided. However, references in local variables may be declared with no explicit lifetime annotation, in which case the lifetimes have to be inferred. This inference makes the validation of the lifetimes more complex. One example is the function `f2`.

```
1  fn f2<'a, 'b>(x: &'a i32) -> &'a i32 {
2      let p = x;       // p -> 'a
3      let q = p;       // q -> 'a
4      return p;
5  }
```

In the second line, the variable `p` is declared with no lifetime annotation. To ensure that `p` is handled properly and that no rules are broken, we need to infer the lifetime of `p`, by propagating information about lifetimes on which `p` depends. Recall the rule that states that a reference cannot live longer than the value it points to. By applying this rule to the first assignment in `f2`, we deduce that the lifetime of `p` cannot outlive the lifetime of `x` (`'a`). However, we want to infer the longest possible lifetime, i.e., the "strongest postcondition", and thus we determine that the lifetime of `p` is `'a`. Extending the same reasoning to the second assignment, we also infer that `q` also has lifetime `'a`.

When we try to return `p`, the code is valid because it was inferred that `p` also has lifetime `'a`. However, what happens if `p` depends on more than one variable?

```
1  fn f3<'a, 'b>(x: &'a i32, y: &'b i32) -> &'b i32 {
2      let mut p = x;       // p -> 'a
3      p = y;               // p -> 'b ?
4      return p;
5  }
```

Function `f3` illustrates this case. One might expect that the assignment on line 3 updates the lifetime of `p` to `'b` and, for that reason, the code would be valid (note that the lifetime of the return value of `f3` is `'b`). However, Rust's lifetime analysis defines that the lifetime of `p` is not `'b`. Instead, it is the intersection between lifetimes `'a` and `'b`. Since the analysis cannot determine if `'b` is shorter than `'a`, the return on line 4 generates an error. To correct `f3`, we would need to annotate both parameters and the return value with the same lifetime (for example, `'a`).

The lifetime analysis in Rust is flow-insensitive and intra-procedural, which means that each function is analyzed independently of the others and that each function has only one state. Therefore, the lifetime of a reference does not change throughout the function: Rust deduces a single lifetime for each reference, in the scope of the function. The analysis' state stores the lifetime constraints on references, allowing the inference of a single lifetime for each reference.

The only way to ensure that the assignments on lines 2 and 3 of `f3` are both correct is by inferring that the lifetime of `p` is no longer than `'a` and no longer than `'b`, thus the lifetime of `p` is the intersection between the lifetimes it depends on.

Function `f4`, defined below, is another example that highlights the flow-insensitive nature of Rust's lifetime analysis:

```
1  fn f4<'a, 'b>(x: &'a i32, y: &'b i32, b: bool) -> &'a i32 {
2      let mut p = x;      // p -> 'a ?
3      if b {
4          return p;
5      }
6      p = y;              // p -> shortest('a, 'b)
7      return p;
8  }
```

One might expect that only the `return` on line 7 would generate an error, since it is not possible that, in the same execution, lines 4 and 6 are both reached. In other words, either the function returns at line 4 and `p` never depends on `q`, or the assignment in line 6 is executed meaning that the program did not reach line 4. However, since the analysis is flow-insensitive and there is only one state per function, the following error will also be generated in line 4:

```
error: lifetime may not live long enough
 --> src/lib.rs:4:16
  |
1 | fn f4<'a, 'b>(x: &'a i32, y: &'b i32, b: bool) -> &'a i32 {
  |       --  -- lifetime `'b` defined here
  |       |
  |       lifetime `'a` defined here
...
4 |          return p;
  |                 ^ function was supposed to return data with lifetime `'a`
  |                   but it is returning data with lifetime `'b`
  |
  = help: consider adding the following bound: `'b: 'a`
```

Let us revisit function `f3` to explain the consequences of having only one state per function. However, this time with some new additions to the code. In function `f5`, we declare two variables (`p` and `q`) and return `q` instead of `p`.

```
1  fn f5<'a, 'b>(x: &'a i32, y: &'b i32) -> &'a i32 {
2      let mut p = x;      // p -> 'a
3      let q = p;          // q -> 'a
4      p = y;              // p -> shortest('a, 'b)
5      return q;
6  }
7
```

Compiling `f5` also produces an error similar to the ones we have seen. It states that `q` may have lifetime `'b`. This is because there is one single state for the analysis of this function. Even though the assignment `p = y` comes after the assignment `q = p`, in fact, the lifetime of `p` is always the intersection between `'a` and `'b`. As a consequence, the lifetime of `q` will also be the intersection between `'a` and `'b`.

In Rust, annotating lifetimes in structures is similar to functions. Each reference inside a structure may have a different lifetime and Rust has syntax for attributing different lifetimes to each reference, provided all of them outlive the structure to which they belong. Therefore, Rust's lifetime analysis is field-sensitive.

In the code snippet below, we can see how a structure with references inside can be created, and how to correctly assign distinct lifetimes to each field.

```
struct S<'c, 'd> {
    f1: &'c u32,
    f2: &'d u32,
}

fn f6<'a, 'b>(v1: &'a u32, v2: &'b u32) -> S<'a, 'b> {
    let s = S {
        f1: v1,
        f2: v2,
    };
    s
}
```

It is to note, however, that arrays in Rust are fixed-sized data structures where all elements share the same type and lifetime.

In summary, there are three important aspects to retain from examining Rust's lifetime analysis:

- The analysis is flow-insensitive, which means that validating lifetimes should come after propagating all lifetimes between variables
- The analysis is intra-procedural and thus we can analyze each function independently. Each function has access to the other functions' signatures but knows nothing about their body.
- The analysis is field-sensitive, meaning that each field in a structure may have a distinct lifetime.

5.2 Lifetime analysis in C++: foundation

In this section, we describe the foundation of the static analysis of our work. In other words, we introduce a simpler version of our solution, which assumes that pointers may only have one level of indirection. Then, in Section 5.3, we delve into more intricate parts of the analysis, which add an additional layer of complexity to what is described in this section.

Our C/C++ lifetime analysis is flow-insensitive and intra-procedural, analogous to Rust's lifetime analysis, for several reasons. Firstly, if Rust's lifetime analysis has these characteristics, then they are sufficient to effectively express a non-trivial set of programs. Secondly, this addresses the requirement for

LSA to be fast and minimize memory overhead. As explained in Chapter 3, a flow-insensitive and intra-procedural analysis is less precise, but performs better in terms of time and memory overhead, when compared to a flow-sensitive and inter-procedural alternative. Additionally, adopting these characteristics makes the research and implementation process more tractable. Finally, our C/C++ lifetime analysis should yield the same results as Rust's. As a consequence, our approach should also consider a single state per function and analyze each function individually and independently of the context.

On the other hand, contrary to Rust's, our solution is field-insensitive as it significantly simplifies the analysis. This means that, regarding arrays, all elements share the same lifetime. Likewise, within structs, all fields share the same lifetime, which is also the lifetime of the struct itself.

It is worth noting that while the current approach is intentionally more straightforward, making the analysis field-sensitive will significantly increase its precision and thus we leave it as interesting future work.

To define the C/C++ lifetime analysis, first, we need to establish its abstract domain. Since we are dealing with lifetimes, the abstract domain is $\{\texttt{\$local}, \$\ell, \top, \bot\}$, where ℓ is a letter that represents an arbitrary lifetime, usually between letters `a` and `z`. These lifetimes alone hold no particular value but allow us to establish relations between the lifetimes of different variables, as explained in Section 2.2.

Then, there are three special lifetimes. The `$local` is the shortest possible lifetime that an accessible value may have. It is therefore always shorter than `$a`, `$b`, and so on. A value whose lifetime is `$local` may be used inside of the respective function's scope but may not be the return value. The `$local` does not explicitly exist in Rust, i.e., we cannot annotate a value with `'local`, but variables may be inferred to have this lifetime. The $\top$ represents the `$static` lifetime. Contrary to lifetime `$local`, `$static` is a special lifetime that denotes that the affected value lives for the entire duration of the program execution, being the largest lifetime possible. The equivalent `'static` lifetime exists in Rust and behaves the same way in C/C++. Note that global variables in C/C++ are variables that exist during the whole program and, for that reason, a variable that is declared as global is automatically attributed a `$static` lifetime, needless of an explicit annotation. Finally, the $\bot$ represents the lifetime `$dead`, which is automatically attributed to values that are not accessible, even inside the function's scope, and it is shorter than `$local`. The concept of lifetime `$dead` does not exist in Rust and its usage in C/C++ is explained better in Section 5.3. From this set of lifetimes, only `$static` and a specifically named lifetime $\$\ell$ may be explicitly used in a lifetime annotation, whereas the lifetimes `$local` and `$dead` can only be inferred during the analysis. We can organize the abstract values of the domain in the lattice from Figure 5.1.

Our analysis also needs a state, which is a valuation of each variable in that function's scope. Since the analysis is flow-insensitive and intra-procedural, each function has a single state and that state is independent of the state of all other functions. When analyzing a function call, the caller has access to the callee's signature, but not to its body or state.

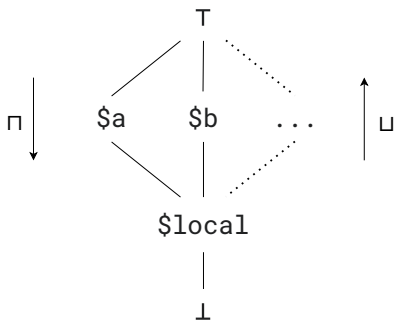

Figure 5.1: C/C++ lifetime analysis lattice.

The state holds, for each function, the following information:

- L: $v \to \mathcal{L} \cup \{\mathsf{nil}\}$
- D: $v \to 2^v$
- R: $\mathcal{L} \cup \{\mathsf{nil}\}$

where v is an arbitrary variable and $\mathcal{L}$ stands for Lifetime, which may be `$local`, `$static`, `$dead` or an arbitrary `$`ℓ (such as `$a`, `$b` and so on). In other words, $\mathcal{L}$ is the set of possible values in the abstract domain. The state holds the lifetime of each variable in the function (L), which may be one of the possible values of $\mathcal{L}$ or nil . L is set for the variables that were declared with a lifetime annotation and is nil for all others. Additionally, the state holds the lifetime dependencies of each variable (D). These are essential to help infer the lifetime of a variable v that was not declared with a lifetime annotation and, consequently, $\mathrm{L}(v) = \mathsf{nil}$. During the analysis, the lifetime of variables with no lifetime annotation is inferred based on the relation to other variables, which create outliving dependencies and are stored in D. The lifetime of the variable is then calculated by finding the minimum of its possible lifetimes, inferred from D. Finally, the state also holds the lifetime of the return value of the current function (R) which, similarly to L, may be $\mathcal{L}$ or nil , in case the function does not return a pointer. If the return value is a pointer/reference, and the function definition does not include a lifetime annotation for the return value, then R holds the lifetime `$local`.

Let us take a look at an example:

```
1  int *$a fn(int *$a x, int *$b y) {
2      int *$b z = y;
3      int *p = x;
4      p = z;
5      return p;
6  }
```

The lifetime algorithm starts by retrieving information from the function signature. It stores the lifetime

annotations for the parameters in L, associating `x` with `$a` and `y` with `$b`. In addition, it also stores the lifetime of the return value in R, which is `$a`.

After that, the algorithm analyzes the function body, going through the code's AST. For each variable declaration, it stores the corresponding lifetime annotation in L, when it exists, and, if not, initializes the dependencies (D) for that variable. So, for the declaration of variable `z`, it stores `$b` in L. In contrast, the declaration of `p` includes no lifetime annotation, which means that the value stored in L is `nil` . Its dependencies vector is initialized and the variable `x` is stored in that vector since it is initializing `p`. Additionally, for each value assigned to a variable whose L is `nil` , the algorithm stores that value in the variable's dependencies vector. Therefore, on the fourth line, the analysis adds `z` to the dependencies vector of `p`, resulting in {`x`, `z`}. Finally, the return statement does not include any valuable information for the propagation of lifetimes and thus is ignored in this step of the analysis.

In summary, the lifetime propagation algorithm computes the following information for the example above:

- $\mathrm{L}_{fn}(x)$ = `$a`, $\mathrm{L}_{fn}(y)$ = `$b`
- $\mathrm{L}_{fn}(z)$ = `$b`, $\mathrm{D}_{fn}(z) = \emptyset$
- $\mathrm{L}_{fn}(p)$ = `nil` , $\mathrm{D}_{fn}(p)$ = {`x`, `z`}
- R_{fn} = `$a`

The L_{fn}, D_{fn} and R_{fn} refer to the information stored in the state of the function `fn`.

Recall that the outlive condition is not implemented in the C++ lifetime annotations and thus we cannot infer whether `$a` outlives `$b` or the opposite, for instance. Therefore, if a variable has the possible lifetimes {`$a`, `$b`, `$local`}, we can infer that its lifetime is `$local`. On the other hand, if the possible lifetimes are {`$a`, `$b`, `$static`}, we can only say that the lifetime of that variable is the intersection between the shortest lifetimes `$a` and `$b`. It is the case of variable `p` in the example above: we can only infer that its lifetime is the intersection between the lifetimes of `x` (`$a`) and `z` (`$b`).

We have described some characteristics of our C/C++ lifetime analysis, as well as defined its abstract domain, lattice, and state. The next step is defining the typing rules that specify the analysis. The notation used in the following rules was introduced in Section 3.4. We have organized the rules into different subsections to make them easier to follow.

5.2.1 Lifetimes

The following rules describe how the lifetimes for each variable are computed, from the information stored in the state. The syntax $\mathsf{baseObj}(e)$, where e is an expression, returns the base pointer of the expression e. For instance, $\mathsf{baseObj}(p + i) = p$ and $\mathsf{baseObj}(p + (q - r)) = p$, where p, q and r are pointers,

and i is an integer. To simplify the typing rules, we assume that $\mathrm{L}_f(e)$ and $\mathrm{D}_f(e)$ apply $\mathsf{baseObj}(e)$. Therefore, $\mathrm{L}_f(p + (q - r))$ returns the lifetime of p and $\mathrm{D}_f(p + (q - r))$ returns the dependencies of p.

$$\frac{\mathrm{L}_f(p)\text{: } \$l}{p : \$l} \quad (\textsc{VarL}) \qquad \frac{\mathrm{L}_f(p)\text{: nil} \qquad \mathrm{D}_f(p)\text{: } \{x_1, \ldots, x_n\}}{p : min\{\mathrm{L}_f(x_1), \ldots, \mathrm{L}_f(x_n)\}} \quad (\textsc{VarNil})$$

$$\frac{g \text{ is global var}}{g : \$static} \quad (\textsc{VarStatic}) \qquad \frac{p \text{ is function pointer}}{p : \$static} \quad (\textsc{FunctionPointer})$$

$$\frac{f : (\ldots,\ \texttt{int*}\ \$l\ x_i,\ \ldots) \to \tau}{\mathrm{L}_f(x_i)\text{: } \$l} \quad (\textsc{ParamDeclL}) \qquad \frac{f : (\ldots,\ \texttt{int*}\ x_i,\ \ldots) \to \tau}{\mathrm{L}_f(x_i)\text{: } \$local} \quad (\textsc{ParamDeclNil})$$

$$\frac{f : (\ldots,\ \texttt{int*}\ \$l\ x_i,\ \ldots) \to \$l\ \tau}{\mathrm{R}_f\text{: } \$l} \quad (\textsc{ReturnDeclL}) \qquad \frac{f : (\ldots) \to \tau}{\mathrm{R}_f\text{: } \$local} \quad (\textsc{ReturnDeclNil})$$

In the current analysis, function pointers are attributed lifetime `$static`, as described by the FUNCTIONPOINTER rule.

5.2.2 Expressions

In this subsection, we present the typing rules for expressions, and the lifetimes associated with each expression.

$$\frac{e \text{ is an integer expression} \qquad \mathrm{L}_f(p)\text{: } \$l \qquad \mathsf{op} \in \{+,\ -\}}{(p\ \mathsf{op}\ e) : \$l} \quad (\textsc{PtrArith})$$

$$\frac{\mathrm{L}_f(p)\text{: } \$a \qquad \mathrm{L}_f(q)\text{: } \$b}{(c\ ?\ p\ :\ q) : min\{\$a, \$b\}} \quad (\textsc{CondOp})$$

$$\frac{S \text{ is an expression of struct type} \qquad \mathrm{L}_f(S)\text{: } \$l}{S.x : \$l} \quad (\textsc{MemberExpr})$$

$$\frac{\mathrm{L}_f(p)\text{: } \$l \qquad \$l \neq \$static}{\&p : \$local} \quad (\textsc{AddrOf}) \qquad \frac{\mathrm{L}_f(p)\text{: } \$static}{\&p : \$static} \quad (\textsc{AddrOfStatic})$$

$$\frac{g : (\ldots,\ \texttt{int*}\ \$a\ x_i, \ldots) \to \$a\ \tau \qquad \mathrm{L}_f(p_i)\text{: } \$b}{g(\ldots,\ p_i, \ldots) : \$b} \quad (\textsc{CallExprL})$$

$$\frac{g : (\ldots) \to \$static\ \tau}{g(\ldots) : \$static} \quad (\textsc{CallExprStatic}) \qquad \frac{g : (\ldots) \to\ \tau}{g(\ldots) : \$local} \quad (\textsc{CallExprNil})$$

The MEMBEREXPR rule demonstrates that the analysis is field-insensitive, as described above.

The & operator attributes the `$local` lifetime to most variables (ADDROF). Therefore, their memory

address is accessible throughout the function's body, but cannot be returned since it would reference a variable that is no longer valid. Recall the fundamental rule that a variable may only point to another variable with a longer lifetime. The exception is variables annotated with `$static` (ADDROF$_{\text{STATIC}}$). Since they can live throughout the entire program, then it is valid to access their address at any point in the program and thus the lifetime of the address is also `$static`.

5.2.3 Statements

The following rules define how statements are handled in our C/C++ lifetime analysis. We assume that $\texttt{inst}(f)$ represents the set of instructions inside the function body of f. Since the analysis is intra-procedural, $\texttt{inst}(f)$ may be in any random order. Similarly, $\texttt{expr}(f)$ is the set of expressions inside f and these may also be in an arbitrary order.

$$\frac{(\texttt{int*}\ \$l\ p) \in \texttt{inst}(f)}{\mathrm{L}_f(p)\text{: }\$l} \quad (\textsc{VarDecl}_{\textsc{L}}) \qquad \frac{(\texttt{int*}\ p) \in \texttt{inst}(f)}{\mathrm{L}_f(p)\text{: nil}} \quad (\textsc{VarDecl}_{\textsc{Nil}})$$

$$\frac{(p = q) \in \texttt{inst}(f) \qquad \mathrm{L}_f(p)\text{: nil}}{\text{baseObj}(q) \in \mathrm{D}_f(p)} \quad (\textsc{Assign})$$

If a variable is declared with an explicit lifetime annotation, its lifetime becomes fixed on that value (VARDECL$_{\text{L}}$). On the other hand, if the variable is declared without any annotation, then initially it has no lifetime (VARDECL$_{\text{NIL}}$). Instead, the lifetime of each variable is inferred based on the lifetimes of the variables on which it depends, as explained previously.

It is worth noting that we do not include variable declarations with an initializer in the typing rules, both in this section and in Section 5.3. This is because we assume that variable definitions are equivalent to a variable declaration followed by an assignment.

5.2.4 Errors

The following rules specify situations where LSA generates an error.

$$\frac{(p = q) \in \texttt{inst}(f) \qquad \mathrm{L}_f(p)\text{: }\$a \qquad \mathrm{L}_f(q)\text{: }\$b \qquad \$a > \$b}{Error\text{ : ``assignment requires that `}\$b\text{' outlives `}\$a\text{'"}} \quad (\textsc{Assign}_{\textsc{Error}})$$

$$\frac{(return\ p) \in \texttt{inst}(f) \qquad \mathrm{L}_f(p)\text{: }\$a \qquad \mathrm{R}_f\text{: }\$b \qquad \$a < \$b}{Error\text{ : ``function should return data with lifetime `}\$b\text{' but it is returning data with lifetime `}\$a\text{'"}} \quad (\textsc{Return}_{\textsc{Error}})$$

$$\frac{(return\ p) \in \texttt{inst}(f) \qquad \mathrm{L}_f(p) : \$local}{Error\text{ : ``cannot return data with lifetime `}\$local\text{'"}} \quad (\textsc{Return}_{\textsc{Local}})$$

$$\frac{f : (\$l_1, \ldots, \$l_n) \rightarrow \$l_r \ \tau \qquad \forall_{1 \leq i \leq n} \ \$l_i \neq \$l_r}{Error : \text{“at least one parameter must be annotated with ‘}\$l_i\text{’”}} \quad (\textsc{FuncDecl}_{\textsc{Error}})$$

$$\frac{\begin{array}{cc} g : (x_1, \ \ldots, \ x_n) \rightarrow \tau & \mathrm{L}_f(p_i) \colon \ \$l_i \\ \exists i : \mathrm{L}_g(x_i) = \$static \wedge \$l_i \neq \$static & g(p_1, \ \ldots, \ p_n) \in \texttt{expr}(f) \end{array}}{Error : \text{“argument requires that ‘}\$l_i\text{’ outlives ‘}\$static\text{’”}} \quad (\textsc{CallExpr}_{\textsc{Error}})$$

The rules $\textsc{Assign}_{\textsc{Error}}$ and $\textsc{Return}_{\textsc{Error}}$ essentially follow the fundamental rule that states that a variable cannot reference another variable with a shorter lifetime. The $\textsc{Return}_{\textsc{Local}}$ rule relates to the concept of the `$local` lifetime, explained previously: a value with lifetime `$local` may be used within the function's scope but may not be returned by that function.

Regarding the function declarations, there is only one invalid case: when the lifetime annotation of the return value does not correspond to any of the annotations specified in the parameters ($\textsc{FuncDecl}_{\textsc{Error}}$). The reason behind this is that, as explained in Section 2.2, lifetime annotations by themselves carry no useful information. A lifetime annotation in the return value is only useful if it relates to one or more parameters' lifetime annotations. This relation allows the callee to associate the lifetime of the return value with the lifetime of the corresponding arguments.

Lastly, for this simpler version of the analysis, the function call expression is valid in all situations except when a parameter is annotated with `$static` and the corresponding argument in the callee is not ($\textsc{CallExpr}_{\textsc{Error}}$). This restriction is necessary because all lifetimes are local and relative except for the `$static`, which is global. Consequently, it is invalid to assign to a variable annotated with `$static` a value with a local lifetime and therefore this generates an error. In all other cases, calling a function is valid, in terms of lifetimes.

```
1  void f1(int *$a x, int *$a y) {
2      x = y;
3  }
4
5  void f2(int *$a x, int *$b y) {
6      f1(x, y);
7  }
```

One might think that the above case is invalid since function `f2` calls a function whose parameters have the same lifetime annotation, but with arguments that do not have the same lifetime annotation. However, in fact, there is always a minimum lifetime between `$a` and `$b`, even if the analysis is not capable of deducing it explicitly. Therefore, we assume that the lifetime `$a` in `f1` will be at least as short as both `$a` and `$b` in `f2`, since, in this case, the scope of function `f1` ends before the scope of function `f2`. This is enough to prove that the function call is valid.

It is important to note that, since the analysis is intra-procedural, `f2` does not have access to the body of `f1`, only to the function signature. Essentially, the analysis of `f2` works under the assumption that function `f1` is correct and focuses only on its signature.

5.3 Lifetime analysis in C++: complex rules

In this section, we delve into the more complex aspects of our C/C++ analysis. We start by addressing the specific challenges that justify this additional complexity. Afterward, we introduce the typing rules necessary for accommodating these additions. The foundation rules defined previously serve as a starting point. Some of these rules accurately describe the analysis, while others are redefined below. In the latter cases, we reuse the corresponding rule name. Additionally, new rules are required, which are also presented in this section.

5.3.1 Pointers with Multiple Levels of Indirection

To dive into this topic, first, we need to discuss pointer aliasing: two seemingly different pointers may point to the same location and make arbitrary changes to it. As a consequence, the value of a variable may be changed by modifying another variable's value.

```
1  void f1(int *p1, int *p2, int *p) {
2      p1 = p;
3      p2 = p;
4  }
```

Function `f1` illustrates pointer aliasing in C/C++. Pointers `p1`, `p2` and `p` all point to the same location, and all of them can make changes to it.

In Rust, there can also be multiple references to the same object. However, the mutability rules prevent aliasing from being harmful by stating that there can be either multiple immutable references to a value or exactly one mutable reference, but not both at the same time. Since immutable references are not capable of changing the object that they point to, Rust's lifetime analysis does not need to handle pointer aliasing.

On the other hand, C and C++ both allow pointers to the same location to read and change data arbitrarily. This is one of the causes of unsafety in C and C++, as it leads to data races and undefined behavior. Let us consider the following example:

```
1  void f2(int *$a p1, int *$b p2) {
2      int **p;
3      p = &p1;
4      *p = p2;
5  }
```

After the second assignment (line 4), `*p` and `p2` point to the same location but their addresses are different. However, in the first assignment (line 3), we have that `p` points to the address of variable `p1`, which means that `*p` and `p1` are now the same. Consequently, the assignment `*p = p2` is equivalent to `p1 = p2`, which should generate a warning due to the lifetimes of `p1` and `p2` being different.

For this reason, on the second assignment, it is not enough to check if the lifetime of `p2` is at least as long as the lifetime of `*p`. On top of this, we must ensure that the lifetime of `p2` is not shorter than any possible lifetime of `p`, in this case `p1`. The lifetime of `p2` is `$b` and the lifetime of `p1` is `$a`. Since the analysis cannot ensure that `$a` $\leq$ `$b`, then it should reject function `f2`.

This introduces a new level of complexity that needs to be handled in C and C++. Note that each level of indirection of a variable has a distinct lifetime associated with it. Since Rust's lifetime analysis does not handle pointer aliasing, we had to come up with an original solution that is consistent with the rest of the lifetime analysis. The solution is defined by the following two rules, given an assignment `x = y`:

- For the outer level of indirection, the lifetime of `x` must not outlive the lifetime of the `y`. This rule was already introduced in Section 5.2.
- For each inner level of indirection, the lifetime of `y` must be at least as long as the maximum of the possible lifetimes of `x`, at the same level of indirection.

Let us revisit function `f2` to explain why this second rule is sufficient to ensure memory safety in assignments. In the assignment in line 4, the analysis checks the lifetime of `p2` against all possible lifetimes of `*p`. By doing so, it guarantees that the code is not indirectly modifying any variable whose lifetime is longer than that of `p2`. In other words, the analysis only accepts the assignment if `p2` outlives any alias of `*p`, thereby preventing the creation of dangling pointers.

This approach is described by the assignment rules introduced in Sections 5.3.5 and 5.3.6.

5.3.2 Function calls and lifetime *$dead*

The analysis of function calls becomes much more complex when we introduce pointers with multiple levels of indirection.

```
1  void f1(int **p);
2
3  int* f2(int **x) {
4      f1(x);
5      return *x;
6  }
```

The function call to `f1` on the fourth line may change the pointer `*x` arbitrarily because, while `x` is passed by value to `f1`, `*x` is passed by reference. Consequently, the changes made to `*p` in function `f1` affect `*x` in function `f2`. Given that the analysis is intra-procedural, when analyzing `f2`, the analysis has access to `f1`'s signature but not its body. Therefore, the analysis must assume that, after the function call to `f1`, `*x` may have any lifetime. Note that the inner indirection of `p` in `f1` behaves just like a return value, since function `f2` can make use of `*x` as if it was returned by the function call. Consequently, for

the same reason, it is illegal to return a variable with lifetime `$local`, the C/C++ lifetime analysis must also determine how `*x` can be used in function `f2`.

To handle this, we introduced the concept of a special lifetime `$dead`. It means that the corresponding variable exists but cannot be read from and thus it is the shortest possible lifetime. The lifetime `$dead` enables the analysis to handle the case where we pass a variable as an argument to a function call, but the corresponding parameter has lifetime `$local`. Note that it is not legal to annotate pointers with `$local`, but in our C/C++ lifetime analysis, we assume that a parameter with no explicit annotation has lifetime `$local`. This differs from Rust, and we discuss this in more detail in Section 8.2. This lifetime does not exist in Rust and explicitly annotating a variable with `$dead` is not allowed.

If we want to read `*x` inside `f2`, then we must add a lifetime annotation to `*p` in `f1`'s signature. This way, when analyzing `f1`, the analysis ensures that `*p` can only be assigned a variable whose lifetime is at least as long as `$a`. Then, when analyzing `f2`, the analysis assumes that `f1` is valid and can verify the function call based on that assumption. Our C/C++ lifetime analysis accepts the following function call, in `f4`, because the code is correctly annotated.

```
1  void f3(int *$a *p) {
2      int i = 0;
3      *p = &i;     // warning
4  }
5
6  int *$a f4(int *$a *x) {
7      f3(x);
8      return *x;
9  }
```

While processing `f4`, the analysis has no access to the body of `f3`, but, since the parameter has a lifetime annotation, the analysis assumes that the lifetime of `x` will not be shorter after the function call. The analysis does generate a warning while processing `f3`, because, on the third line, the code assigns `&i`, with lifetime `$local`, to `*p`, with lifetime `$a`.

Let us assume that parameter `p` had no lifetime annotation in function `f3`. The lifetime of `*p` in `f3` would be `$local` by default, thus making the assignment on line 3 valid. Then, inside `f4`, `*x` would be attributed lifetime `$dead` because of the function call on line 7. For this reason, our analysis would generate the following error on line 8:

```
example.cpp:8:13: warning: function should return data with lifetime '*$a'
                           but it is returning data with lifetime '*$dead'
    return *x;
    ~~~~~~~~~
example.cpp:7:5: note: declared with lifetime '*$dead' here
    f3(x);
    ^~~~~
```

In the examples above, the parameter was assigned data local to the function. However, a parameter may also be assigned another parameter, and our lifetime analysis must also handle these cases.

```
1  void f5(int *$a *p, int *$a q) {
2      *p = q;
3  }
4
5  int *$a f6(int *$a *x, int *$b y) {
6      f5(x, y);
7      return *x;
8  }
```

Function `f5` is valid: in the assignment, both `*p` and `q` have lifetime `$a`. However, in function `f6`, after the function call, `*x` points to `y`, which is invalid in this context because there is no guarantee that `$b` is at least as long as `$a`. Recall that while the analysis processes `f6`, it has no access to the body of `f1`, and thus the analysis has to be pessimistic. In this case, it has to assume that, since `*p` and `q` have the same lifetime in `f5`, then `*x` may be assigned `y` inside that function. Note that the other way around may also happen, but since `y` is passed by value, such assignment does not affect `f6`.

To handle function calls with multiple levels of indirections, our C/C++ lifetime analysis must validate the lifetimes between parameters. The rules CALLEXPR$_{\text{ERROREQUAL}}$ and CALLEXPR$_{\text{ERRORSHORTER}}$, in Section 5.3.6, describe this validation. Our analysis would then produce the following error for the example above:

```
example.cpp:6:5: warning: when calling function 'f5', the lifetime of 'y'
                            cannot be shorter than the lifetime of '*x'
    f5(x, y);
    ^~~~~~~~
example.cpp:5:32: note: lifetime is '*$b'
int *$a f6(int *$a *x, int *$b y) {
                       ~~~~~~~~~
example.cpp:5:21: note: lifetime is '*$a'
int *$a f6(int *$a *x, int *$b y) {
           ~~~~~~~~~~
```

Note that this verification is only required because the parameters are mutable. If they were const instead, the callee would assume that the arguments could not be changed during the function call. Consequently, by making `*x` const in `f5`'s signature above, the analysis would no longer generate any warning. Below is another example where the use of the keyword `const` eliminates potential warnings.

```
1  void f7(int * const *p);
2  void f8(int * const *p, int *q);
3
4  int *$a f9(int *$a *x, int *$b y) {
5      f7(x);
6      f8(x, y);
7      return *x;
8  }
```

This code is valid and accepted by the analysis. The function call to `f7` in line 5 would attribute lifetime `$dead` to `x` if the parameter `*p` was not constant. Then, a warning would be generated in line 7,

due to the attempt to return data with lifetime `$dead`. Likewise, if parameter `*p` was not constant in `f8`, the analysis would generate a warning similar to the above.

5.3.3 Lifetimes

The following rules essentially adapt some of the rules from Section 5.2.1 to pointers with multiple levels of indirection. These rules are defined for a single level of indirection. For instance, the PARAMDECL$_{\text{L}}$ rule has to be applied for each level of indirection of the declaration of parameter p, producing a different $\mathrm{L}_f(\overset{n}{*\cdots*}p)$ for each specific level of indirection. The remaining rules from that section are already in accordance with the additional complexity of the analysis.

Recall the syntax $\mathsf{baseObj}(e)$, which returns the base pointer of the expression e. To accommodate pointers with multiple levels of indirection, we define that $\mathsf{baseObj}(\overset{n}{*\cdots*}p) = \overset{n}{*\cdots*}\,\mathsf{baseObj}(p)$. Therefore, $\mathrm{L}_f(\overset{n}{*\cdots*}p)$ returns the lifetime of the n^{th} indirection of p.

$$\frac{g \text{ is global var}}{\overset{n}{*\cdots*}g : \$static} \quad (\text{VAR}_{\text{STATIC}})$$

$$\frac{f : (\ldots,\ \texttt{int}\overset{n}{*\cdots*}\ *\$l\ \overset{m}{*\cdots*}\ x_i,\ \ldots) \to \tau}{\mathrm{L}_f(\overset{m}{*\cdots*}x_i)\colon \$l} \quad (\text{PARAMDECL}_{\text{L}})$$

$$\frac{f : (\ldots,\ \texttt{int}\overset{n}{*\cdots*}\ *\ *\overset{m}{*\cdots*}\ x_i,\ \ldots) \to \tau}{\mathrm{L}_f(\overset{m}{*\cdots*}x_i)\colon \$local} \quad (\text{PARAMDECL}_{\text{NIL}})$$

$$\frac{f : (\ldots,\ \texttt{int}\overset{n}{*\cdots*}\ *\$l\ \overset{m}{*\cdots*}\ x_i,\ \ldots) \to \overset{a}{*\cdots*}\ *\$l\ \overset{b}{*\cdots*}\tau}{\mathrm{R}_f(\overset{b}{*\cdots*})\colon \$l} \quad (\text{RETURNDECL}_{\text{L}})$$

$$\frac{f : (\ldots) \to \overset{n}{*\cdots*}*\ *\overset{m}{*\cdots*}\tau}{\mathrm{R}_f(\overset{m}{*\cdots*})\colon \$local} \quad (\text{RETURNDECL}_{\text{NIL}})$$

The PARAMDECL$_{\text{L}}$ and RETURNDECL$_{\text{L}}$ rules describe how an explicit lifetime annotation for a specific level of indirection is stored in the state. On the other hand, the PARAMDECL$_{\text{NIL}}$ and RETURNDECL$_{\text{NIL}}$ rules handle the cases where a particular level of indirection of a parameter or return value was declared with no lifetime annotation, and thus assigned lifetime `$local` by default.

5.3.4 Expressions

$$\frac{S \text{ is struct} \qquad \mathrm{L}_f(S)\colon \$l}{(*\overbrace{\cdots}^{n})S.x : \$l} \ (\textsc{MemberExpr}) \qquad \frac{\mathrm{L}_f(*\overbrace{\cdots}^{n}p)\colon \$l}{*\overbrace{\cdots}^{n}p : \$l} \ (\textsc{PtrDeref})$$

$$\frac{g : (\dots,\ \texttt{int}*\overbrace{\cdots}^{n}* *\$a *\overbrace{\cdots}^{m}* x_i, \dots) \to *\overbrace{\cdots}^{a}* *\$a *\overbrace{\cdots}^{b}*\tau \qquad \mathrm{L}_f(*\overbrace{\cdots}^{m}*p_i)\colon \$b}{(*\overbrace{\cdots}^{b}*)g(\dots,\ p_i, \dots) : \$b} \ (\textsc{CallExpr}_{\mathrm{L}})$$

$$\frac{g : (\dots,\ \texttt{int}*\overbrace{\cdots}^{n}* * *\overbrace{\cdots}^{m}* x_i, \dots) \to \tau \qquad g(\dots,\ p_i, \dots) \in \texttt{expr}(f) \qquad m > 0}{\mathrm{L}_f(*\overbrace{\cdots}^{m}*p_i)\colon \$dead} \ (\textsc{CallExpr}_{\textsc{Dead}})$$

Since our C++ lifetime analysis is field-insensitive, the MEMBEREXPR rule defines that all levels of indirection of all fields in a struct share the same lifetime as the struct itself. The PTRDEREF rule illustrates that the state stores a different lifetime for each level of indirection of a variable. Finally, the CALLEXPR$_{\text{DEAD}}$ describes the attribution of the lifetime `$dead`, as discussed in Section 5.3.2.

5.3.5 Statements

In this and the following subsection, we assume that α represents the total number of indirections of the type of p, the variable on the left-hand side of the assignment. Note that $n \leq \alpha$ is always true.

$$\frac{(*\overbrace{\cdots}^{n}*p = *\overbrace{\cdots}^{m}*q) \in \texttt{inst}(f)}{\forall i \in [1, \alpha - n], \left(\mathsf{baseObj}(*\overbrace{\cdots}^{m+i}*q) \in \mathrm{D}_f(*\overbrace{\cdots}^{n+i}*p) \land \mathsf{baseObj}(*\overbrace{\cdots}^{n+i}*p) \in \mathrm{D}_f(*\overbrace{\cdots}^{m+i}*q)\right)} \ (\textsc{Assign})$$
$$\mathsf{baseObj}(*\overbrace{\cdots}^{m}*q) \in \mathrm{D}_f(*\overbrace{\cdots}^{n}*p)$$

The ASSIGN rule defines how assignments between pointers with multiple levels of indirection are handled, as described in Section 5.3.1. Essentially, the analysis creates dependencies both ways for all but the outer level of indirection of the pointers. Note that, if the right-hand side `q` is const, then `p` is not included in the dependencies of `q`, for that indirection level.

5.3.6 Errors

In Section 5.2.4, we specified some errors generated by LSA. The following rules specify the additional situations that generate an error, accommodating the complexity introduced in this section.

$$\frac{(*^{\overbrace{\cdots}^{n}}p = *^{\overbrace{\cdots}^{m}}q) \in \texttt{inst}(f) \qquad \mathrm{L}_f(*^{\overbrace{\cdots}^{n}}p)\colon \$a \qquad \mathrm{L}_f(*^{\overbrace{\cdots}^{m}}q)\colon \$b \qquad \$a > \$b}{Error : \text{“assignment requires that ‘}*^{\overbrace{\cdots}^{m}}\$b\text{’ outlives ‘}*^{\overbrace{\cdots}^{n}}\$a\text{’”}} \ (\textsc{Assign}_{\textsc{Error}})$$

$$\frac{\begin{array}{c}(*^{\overbrace{\cdots}^{n}}p = *^{\overbrace{\cdots}^{m}}q) \in \texttt{inst}(f) \qquad max(\mathrm{D}_f(*^{\overbrace{\cdots}^{n+i}}p)) : \$a \\ \mathrm{L}_f(*^{\overbrace{\cdots}^{m+i}}q)\colon \$b \qquad \$a > \$b \qquad i \in [1, \alpha - n]\end{array}}{Error : \text{“assignment requires that ‘}*^{\overbrace{\cdots}^{m+i}}\$b\text{’ outlives ‘}*^{\overbrace{\cdots}^{n+i}}\$a\text{’”}} \ (\textsc{Assign}_{\textsc{ErrorMax}})$$

$$\frac{*^{\overbrace{\cdots}^{n}}p \ \in \texttt{expr}(f) \qquad \mathrm{L}_f(*^{\overbrace{\cdots}^{n}}p)\colon \$dead \qquad n > 0}{Error : \text{“cannot read data with lifetime ‘}*^{\overbrace{\cdots}^{n}}\$dead\text{’”}} \ (\textsc{Dead}_{\textsc{Error}})$$

$$\frac{\begin{array}{c}g : (\ldots,\ \texttt{int}*^{\overbrace{\cdots}^{n}} * \$l\ *^{\overbrace{\cdots}^{m}} * x_i, \ldots,\ \texttt{int}*^{\overbrace{\cdots}^{a}} * \$l\ *^{\overbrace{\cdots}^{b}} * x_j, \ldots) \to \tau \\ \mathrm{L}_f(*^{\overbrace{\cdots}^{m}}p_i)\colon \$a \qquad \mathrm{L}_f(*^{\overbrace{\cdots}^{b}}p_j)\colon \$b \qquad n > 0 \qquad m > 0 \\ g(\ldots,\ p_i,\ \ldots,\ p_j,\ \ldots) \in \texttt{expr}(f)\end{array}}{Error : \text{“when calling function } g\text{, the lifetimes of ‘}*^{\overbrace{\cdots}^{m}}p_i\text{’ and ‘}*^{\overbrace{\cdots}^{b}}p_j\text{’ should be the same”}} \ (\textsc{CallExpr}_{\textsc{ErrorEqual}})$$

$$\frac{\begin{array}{c}g : (\ldots,\ \texttt{int}*^{\overbrace{\cdots}^{n}} * \$l\ *^{\overbrace{\cdots}^{m}} * x_i, \ldots,\ \texttt{int}*\$l\ x_j, \ldots) \to \tau \qquad n > 0 \\ \mathrm{L}_f(*^{\overbrace{\cdots}^{m}}p_i)\colon \$a \qquad \mathrm{L}_f(p_j)\colon \$b \qquad g(\ldots,\ p_i,\ \ldots,\ p_j,\ \ldots) \in \texttt{expr}(f)\end{array}}{\begin{array}{c}Error : \text{“when calling function } g\text{, the lifetime of ‘}p_j\text{’} \\ \text{cannot be shorter than the lifetime of ‘}*^{\overbrace{\cdots}^{m}}p_i\text{’”}\end{array}} \ (\textsc{CallExpr}_{\textsc{ErrorShorter}})$$

$$\frac{\begin{array}{c}g : (\ldots,\ \texttt{int}*^{\overbrace{\cdots}^{n}} * \$l\ *^{\overbrace{\cdots}^{m}} * x_i, \ldots,\ \texttt{int}*^{\overbrace{\cdots}^{a}} * * \$l\ \texttt{const}\ *^{\overbrace{\cdots}^{b}} * x_j, \ldots) \to \tau \\ \mathrm{L}_f(*^{\overbrace{\cdots}^{m}}p_i)\colon \$a \qquad \mathrm{L}_f(*^{\overbrace{\cdots}^{b}}p_j)\colon \$b \qquad n > 0 \qquad m > 0 \\ g(\ldots,\ p_i,\ \ldots,\ p_j,\ \ldots) \in \texttt{expr}(f)\end{array}}{\begin{array}{c}Error : \text{“when calling function } g\text{, the lifetime of ‘}*^{\overbrace{\cdots}^{b}}p_j\text{’} \\ \text{cannot be shorter than the lifetime of ‘}*^{\overbrace{\cdots}^{m}}p_i\text{’”}\end{array}} \ (\textsc{CallExpr}_{\textsc{ErrorConst}})$$

6

Implementation

Contents

In the previous chapter, we delved into the theoretical aspects of our C/C++ lifetime analysis. We defined its characteristics, introduced the rules that specify the analysis, and discussed its limitations and design choices.

Now, we introduce the implementation of LSA.[1] We start by exploring the high-level architecture of LSA and then examine more in-depth how each function is analyzed individually. Before doing so, we introduce Clang and LLVM, on which our implementation is based.

6.1 Clang and LLVM

LLVM [37] is a collection of reusable compiler components, including a set of capabilities helpful to program analysis. It includes a low-level code representation that is reusable and flexible, allowing

[1] GitHub repository: https://github.com/susmonteiro/llvm-project/tree/lifetimes-static-analyzer

the development of compilers for different programming languages. LLVM is currently used to compile languages such as C, C++, Fortran, and Rust, among others.

Clang is a compiler for the C language family, e.g., C, C++, Objective-C, etc. It uses LLVM as backend and was developed by the LLVM community as a free and open-source alternative to other compilers such as GCC. It was released in 2007 and has become a popular choice for C/C++ development.

In addition to Clang, the LLVM project also includes Clang-Tidy, a tool that was developed more recently, on top of Clang. Clang-Tidy allows developers to automatically refactor their code, improving its quality, maintainability, and readability by making use of the latest C++ language features.

Clang includes a tool called the Clang Static Analyzer (CSA).[2] This is a source code bug-finding tool that allows developers to create and integrate their own checkers.

6.2 Lifetimes static analyzer

LSA is implemented directly within Clang's codebase. Alternatively, we could have used the CSA capabilities, since our tool is a static analyzer. However, the CSA analysis is inherently flow-sensitive and inter-procedural, in contrast to our C/C++ analysis, which adopts a flow-insensitive and intra-procedural approach. For this reason, we found little advantage in using the CSA capabilities, which led us to the decision to implement LSA right within Clang.

LSA's implementation comprises approximately 4.2k lines of code. It is written in C++ and uses the Clang AST to perform its analysis on the code. Specifically, whenever Clang analyzes a function, LSA is invoked to perform its verification on that same function. We invoke LSA in two instances of the Clang AST. The first one is on function declarations, i.e., where the function signatures are analyzed, with no access to their bodies. LSA gathers the lifetime annotations of the parameters and the return value of the current function and stores this information in the state, which is global to the whole analysis. This information is crucial when inspecting function calls. Although each function is analyzed individually, when we come across function calls in the body of a function, we require access to the callee's parameters and return lifetimes. Since function calls can happen at any point in the code, we must keep this information throughout the entire analysis. Note, however, that the volume of information stored from the function signatures is insignificant when compared with the data generated during the analysis of a single function body.

The analysis of function signatures mostly involves gathering information. At this point, the analysis has no access to the function body, therefore it only checks whether the lifetime annotation of the return value, if present, is either `$static` or matches at least one of the lifetime annotations in the parameters. The reason for this verification is explained in Section 5.2.4.

[2] https://clang-analyzer.llvm.org/

The second instance where LSA is called is on function definitions, which enables the analysis of the corresponding function body. Most of the C/C++ lifetime analysis is performed at this stage. As discussed previously, there is a single state per function, which means that when validating each statement, we must take the context of the whole function's body. In other words, one statement involving variable `x` may be incorrect due to `x` being set with a specific lifetime after the current statement. Consequently, we must first gather and propagate all important information for the analysis, i.e., the dependencies between variables, and figure out the lifetime of each variable. Only after that can we go through the code again and verify whether the code is valid. For this reason, we split the analysis into the following three distinct steps, requiring LSA to go through each function body twice.

1. Acquisition: gather lifetime information from the entire function body and create a dependency graph between variables.
2. Propagation: propagate the dependencies collected.
3. Validation: verify if the code is correct, using the information collected in the previous two steps, and generate the necessary warnings.

The acquisition and validation steps both require traversing the Clang AST, with the propagation step being the only one independent of it.

We will delve into each of these steps in detail in the following subsections. To help explain them, we will use the following example function:

```
1  int *$b fn(int *$a p) {
2      int *x = p;
3      int *y = x;
4      return y;
5  }
```

Note that, before analyzing the body of the function, LSA generates the following warning when processing the function's signature:

```
example.cpp:1:9: warning: at least one parameter must be annotated with '$b'
int *$b fn(int *$a p) {
~~~~~~~~~~
```

6.2.1 Acquisition

The first step in the analysis is acquisition. It involves going through the code and gathering the lifetime of each variable. For annotated variable declarations, this involves simply associating the variable with the defined lifetime. In all other cases, we look for dependencies between the lifetimes of different variables and store those dependencies in a graph structure. These dependencies are used in the second step to

infer the set of possible lifetimes for each variable declared with no lifetime annotation. The acquisition step corresponds to the first iteration through the code's AST.

After performing the first step in the code snippet above, this is the information collected:

```
p : $a
x -> p
y -> x
```

The parameter `p` is declared with lifetime `$a`, which is stored in the state. In the second line, we have a variable definition of pointer type with no lifetime annotation. Consequently, we add the initializer (variable `p`) to the dependencies of `x`. Similarly, on the third line we add `x` to the dependencies of `y`. Finally, the `return` statement in the fourth line provides no information and thus is ignored in this step.

The acquisition step is implemented using the *Visitor* design pattern, used to recursively traverse the AST nodes. We need to handle each type of node differently, depending on whether we need to gather information from it or we may discard it. For instance, assignments, function calls, and variable declarations need to be cautiously analyzed, since they tell us how the variables depend on each other, regarding lifetimes. For these statements, LSA starts by visiting each of the current node's children. Then, it performs the node's specific analysis and stores the relevant information in the state, which will be used both in the acquisition and validation steps. In the acquisition step, the stored information is crucial for the analysis of parent nodes. In the validation step, as explained previously, this information is used to process the code and generate the necessary warnings.

It is also important to analyze expression nodes such as the unary operator `&`, member expressions, and array subscripts to store, in the state, the lifetime corresponding to each of these expressions, based on the typing rules from Sections 5.2.2 and 5.3.4.

For all other nodes, such as if statements and loops, we visit their children but do not compute anything else, as it does not bring valuable information to the acquisition step of the analysis. This behavior is implemented in general visit functions, the `VisitStmt` and `VisitExpr`, which simply visit each of the current node's children, without performing any specific analysis, to continue the recursion.

6.2.2 Propagation

The second step consists of iteratively propagating the dependencies graph until there are no more changes, forming a transitive closure of the dependencies. We employ a fixed-point, simple work-list algorithm to efficiently propagate the dependencies between variables.

After this step, each variable in the dependency graph will have a set of possible lifetimes. The actual lifetime of a variable corresponds to the minimum among that set. For instance, if variable `x` is inferred to have the set of possible lifetimes {`$a`, `$b`, `$local`}, then it is guaranteed that the lifetime of `x` is `$local`. On the other hand, if the set of possible lifetimes is {`$a`, `$b`, `$static`} instead, we can only assume that

its lifetime is min(`$a`, `$b`). When the propagation step is over, the information collected from the function `fn` is:

```
p : $a
x -> p
y -> x, p
```

Essentially, after this step, we propagated the dependency `y -> x` obtained from the acquisition step, ending up with `y -> x, p`. The algorithm stops at this point because there is no more information to be propagated: it has reached a fixed point.

This information can now be used to compute, for each variable that was declared with no lifetime annotation, the set of shortest lifetimes and infer their lifetime. Since `x` does not have a lifetime annotation and it depends on `p`, then its lifetime will be the lifetime of `p`, which is `$a`. Variable `y` depends on `x` and `p`, both with lifetime `$a`. Therefore, the lifetime of `y` is also inferred to be `$a`.

6.2.3 Validation

Finally, in the validation step, the information inferred in the previous two steps is used to check if the code is valid. For instance, it is important to check that, in assignments, the lifetime of the right side of the operation is always at least as long as the left side. When a broken rule is found, LSA generates a warning that helps the user understand the context and how to fix the warning, hopefully improving the correction process.

When performing the validation step on the function `fn`, LSA generates the following warning:

```
example.cpp:4:12: warning: function should return data with lifetime '*$b'
                           but it is returning data with lifetime '*$a'
    return y;
    ~~~~~~~~
example.cpp:3:5: note: declared with lifetime '*$a' here
    int *y = x;
    ^~~~~~~~~~
```

Even though `y` does not have a lifetime annotation and does not depend directly on `p`, after the propagation step, LSA has inferred that the lifetime of `y` is `$a`. Hence the warning stating that the function should return data with lifetime `$b` but instead, it is returning `y`, with lifetime `$a`.

The validation's implementation is similar to the acquisition step since it also employs the *Visitor* design pattern to traverse Clang's AST. However, the implementation of each node is more complex than the acquisition step. At this point, LSA must analyze all nodes where a lifetime rule may be broken, as defined in Sections 5.2.4 and 5.3.6. Examples of these are assignments, function calls, variable declarations, and return statements.

Note that traversing the AST twice is sufficient due to the propagation step of the analysis. During the first traversal, LSA collects direct dependencies between variables. By computing the transitive closure

of these dependencies, it is guaranteed that a subsequent iteration through the AST does not yield new information. Consequently, the second traversal is only required to validate the code.

6.3 Data structures

In this section, we explore the implementation's data structures to provide insight into the complexity and organization of LSA.

Every lifetime in a program is stored in a structure called `Lifetime`. This is the fundamental structure of our implementation. Recall that each level of indirection of a variable has a different lifetime and thus each `Lifetime` stores the information of one level of indirection of a variable. The set of lifetimes corresponding to the same variable is stored in `ObjectLifetimes`. This object holds a vector, $PointeeObjects$, where each element is a `Lifetime`. The element in the i^{th} position of $PointeeObjects$ is the `Lifetime` for the i^{th} level of indirection of that variable. The `Lifetime` structure holds the following information:

- Id : `char`
- $Dependencies$: `clang::Stmt[`v`][`$\mathcal{L}$`]`
- $ShortestLifetimes$: `int`

where v is an arbitrary variable and, as defined in Section 5.2, $\mathcal{L}$ = {`$local`, `$static`, `$dead`, `$`ℓ}. Id holds the actual lifetime value. It may be an element of $\mathcal{L}$ or `NOTSET`, in case the variable was not declared with a lifetime annotation. When the Id is `NOTSET`, the lifetime must be inferred from the dependencies between variables, as explained previously. These are stored in $Dependencies$, which is a vector that holds the statements on which the variable depends. Within $Dependencies$, each position corresponds to a specific value of $\mathcal{L}$. For instance, position 1 corresponds to `$static`, and position 2 corresponds to `$a`. Each of these positions holds a vector of the dependencies related to that specific value. If `Lifetime` does not depend on `$local`, for instance, then $Dependencies$ will hold an empty vector for the corresponding position of `$local`. Note that, for validating the code, it would be enough to store a simple vector with the involved values of $\mathcal{L}$, disregarding the set of statements for each of those values. However, storing the `clang::Stmt` is very important to enable LSA to provide the most informative error messages possible. Finally, $ShortestLifetimes$ is a binary mask that, when applied to $Dependencies$, results in the set of shortest lifetimes for that object.[3] So, for instance, if `Lifetime` depends on `$a`, `$b` and `$static`, then its mask will be 1 in the positions corresponding to `$a` and `$b`, and 0 for all other positions. Most times, it is enough to have only the set of shortest lifetimes for each `Lifetime`. However, when validating assignments with multiple levels of indirection, the analysis needs

[3] In the LSA, we assume that arbitrary lifetimes range from `$a` to `$z`, which vastly exceeds the amount required in practice.

to have the maximum of the possible lifetimes for `Lifetime`, which would be `$static` in this case. The binary mask $ShortestLifetimes$ efficiently stores this information without duplicating the data.

Let us take a look at an example.

```
1  void f1(int *$a x, int *$static y) {
2      int *p = x;
3      p = y;
4  }
```

Id is `$a` and `$static` for `x` and `y`, respectively. For variable `p`, Id is initially `NOTSET`. After the acquisition step of LSA, $Dependencies$ of `p` will be `[[], [p = y], [int *p = x]]`. The empty vector corresponds to the lifetime `$local`, on which `p` does not depend. The following two vectors correspond to lifetimes `$static` and `$a`, respectively.

However, in fact, the lifetime of `p` depends only on variable `x`, because `$a` $<$ `$static`. This is stored in the mask $ShortestLifetimes$. After the propagation step, all lifetimes are processed and, if possible, the Id is set to a value of $\mathcal{L}$. In the case of the variable `p`, Id is set to `$a`, since it is the shortest of its possible lifetimes.

`Lifetime` and `ObjectLifetimes` are the two fundamental structures of our implementation. In Figure 6.1 we present an UML diagram to describe LSA's overall structure.

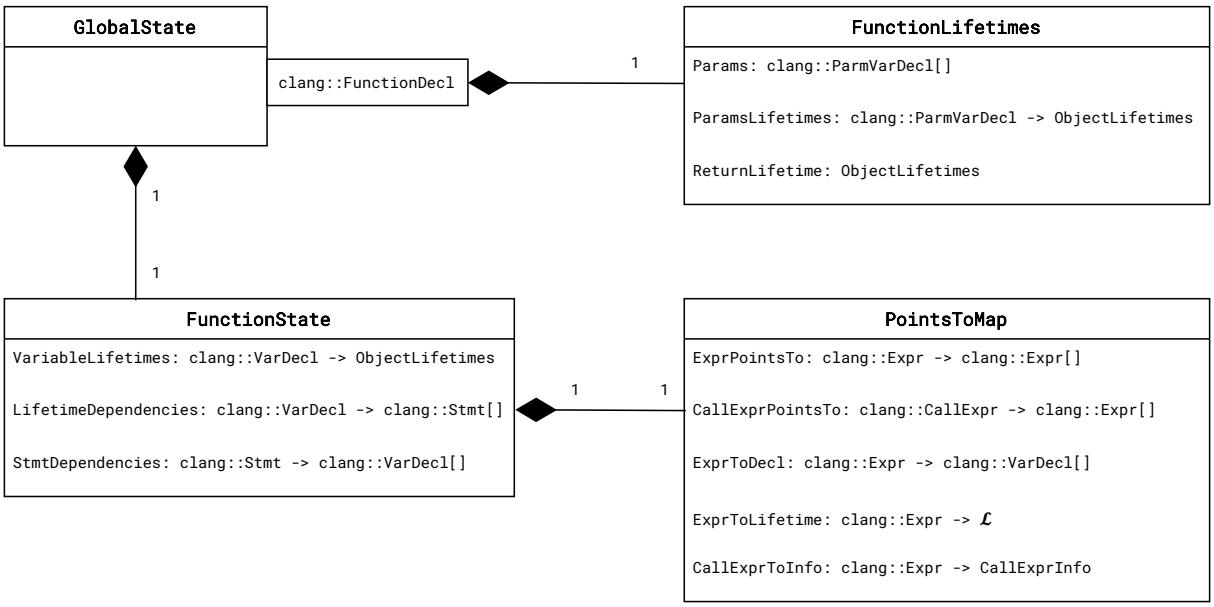

Figure 6.1: High-level structure of LSA.

The `GlobalState` class holds the information that is stored throughout the whole analysis. It contains a map that stores the information about the lifetimes collected from all function signatures. For each `clang::FunctionDecl` in that map, there is an object of type `FunctionLifetimes` that stores the lifetime of the parameters and the return value for that specific function.

The class `FunctionLifetimes` stores a vector of the parameters of its corresponding function, a map that associates each parameter with the respective `ObjectLifetimes` object and an additional `ObjectLifetimes` object to store the lifetime of the return value. The vector $Params$ is useful for knowing the index of each parameter when we need to associate an argument with the parameter, for instance.

Additionally, `GlobalState` also stores an object of type `FunctionState`, that holds the state of the current function body analysis. When LSA finishes analyzing a function body, the object of type `FunctionState` is deleted and a new one is created for the next function body. `FunctionState` contains the map $VariableLifetimes$ with stores, for each variable declared in the function body, the corresponding `ObjectLifetimes` object. The $LifetimeDependencies$ and $StmtDependencies$ structures are used to store information during the analysis that might prove useful when generating error messages. Together, they allow the analysis to determine that a variable `x` depends on a variable `y` through a specific statement that is common between the two structures. This information is gathered during the acquisition step and is used during the propagation step, to fill up the $Dependencies$ structure inside `Lifetime`.

Finally, `FunctionState` also stores a `PointsToMap` object. This structure stores crucial information for the recursive traverse through Clang's AST. When visiting the leaves of the AST, there is important information that needs to be propagated to the parents, which is stored in the data structures of `PointsToMap`. The $ExprPointsTo$, $CallExprPointsTo$, and $ExprToDecl$ structures store, respectively, the expressions and variable declarations to which general expressions and function calls point. These dependencies are stored while traversing the AST during the acquisition step and they are useful, for example, when validating assignments. For instance, suppose that we have pointers `x` and `y` and we perform the assignment `x = y + 1`. In this case, $ExprToDecl$ has the information that the expression `y + 1` depends on `y`, thus preventing a new traversal of the AST and simplifying the validation process.

The structure $ExprToLifetime$ holds the lifetime of specific expressions, to prevent their lifetime from being computed several times. For instance, it stores that the lifetime of `&v` is `$local` or `$static`, depending on the lifetime of `v`. Finally, $CallExprToInfo$ is a map that, for each expression, stores information about the function call whose return value it depends on. Consider the following example:

```
1  int *$a f2(int *$a x, int *$a *y);
2
3  void f3(int *$a p, int *$a *q) {
4      int *x = f2(p, q);
5  }
```

The lifetime of `x` in `f3` depends on the lifetimes of `p` and `*q`, because of the information the analysis extracts from the signature of `f2`. This information is obtained when analyzing the function call `f2(p, q)` and must be propagated to its parents, so that the dependencies of `x` are correctly created. This is the type of information stored in the `CallExprInfo` structure.

7

Evaluation

Contents

To evaluate LSA, we developed a script that generates random dummy C code.[1] This script allows us to create tests with an arbitrary number of lines and specific characteristics, which is very useful to study the performance of LSA. We discuss the results of these tests in Section 7.1.

Additionally, we applied LSA to two benchmarks: Bzip2[2] and SQLite3.[3] In Section 7.2, we present these two benchmarks, comparing and discussing the performance of LSA on them.

7.1 Load tests

The dummy C code generator is a script, written in Python, designed to produce C files with a specific set of characteristics. Its goal is to assess the performance of LSA on large C files, which do not need to compute any practical operation other than successfully compile.

[1] GitHub repository: https://github.com/susmonteiro/dummy-c-code-generator
[2] http://www.bzip.org/
[3] https://www.sqlite.org/

The script creates a single, very large function. Inside the body of this function, it generates variable declarations, both with and without an initializer, assignments, if-else conditions, while loops, function calls, and return statements. These constructs do not fully cover the entire spectrum of the C language, but they consist of the most relevant ones in the context of our C/C++ lifetime analysis. In other words, these are the statements responsible for creating dependencies between variables and that are the most interesting for the analysis, allowing LSA to identify and report the complete set of warnings implemented. Additionally, the script generates multiple function declarations, with no body, which are used in the function calls inside the main function.

Note that the script is considered a C code generator because it does not contain any elements that are unique to C++. Nevertheless, the output file is compatible with both the C and C++ programming languages.

The script allows the specification of multiple attributes for the output file, such as the number of lines of code and the number of variable declarations. The load tests were executed on files with the following characteristics:

- Number of different lifetimes: for instance, if the number of lifetimes is 2, then each lifetime annotation may be `$a`, `$b` or `$static`. The default is a random number between 3 and 8.
- Maximum number of indirections: this value was set to 3 for the evaluation, meaning that each pointer may have up to 3 levels of indirection.
- Lines of code: this is the total number of lines in the file. Its default is 50,000 lines.
- Percentage of function declarations: this is a percentage of the lines of code. In the tests conducted, this value was fixed to 2%.
- Percentage of each statement type: the default is 20% for variable declarations, 36% for assignments, 8% for if-else conditions, 8% for while loops, and 28% for function calls. The probabilities for if-else conditions and while loops are lower because, since the analysis is flow-insensitive, their impact on the performance of the analysis is insignificant.
- Percentage of variable declarations with initializer: the default is 30% of all variable declarations
- Percentage of assignments with function calls: the script creates simple C code, and, for that reason, function calls can only appear on their own or on the right-hand side of function declarations. By default, 50% of assignments include a function call.

Using these specifications, the script starts by selecting the set of possible lifetime annotations. Then, it creates all the function headers and, finally, the main function. Note that this script creates large but very simple programs. Therefore, the only types that exist are `void` and `int`.

For each function header, the script randomly selects the type and number of indirections for the return value. Then it randomly chooses the number of parameters, between 0 and 8, as well as the number of indirections for each of the parameters and the corresponding lifetime annotation, or its absence.

The main function header is generated the same way. Inside the body of the main function, the script starts by creating all the variable declarations, which may be initialized with one of the previously declared variables. Similarly to parameters and return values, for each variable, the script randomly determines the number of indirections and corresponding lifetime annotation. The rest of the function is generated based on the probability of the statements until the expected number of lines of code has been reached. If-else and while loop statements can have an arbitrary number of other statements inside and assignments may have function calls on their right-hand side. It is worth noting that all generated code adheres to the rules of the C and C++ programming language, otherwise, it would not compile. For instance, when generating a function call, the script must ensure that the number of arguments aligns with the function's parameters and that the number of indirections for each argument is consistent with the corresponding parameter. Finally, the function may return one of its variables or parameters.

We used the script to generate multiple different random tests, with distinct characteristics. Then, we proceeded to compare the compile and LSA time on each one of them, to assess the performance of our static analyzer. Note that LSA is executed during compilation with the `-O0` flag and, for that reason, the LSA time includes both the compile and analysis time. Additionally, we also recorded the number of generated warnings to study how it varies across different file configurations.

The initial phase of the evaluation involved studying how the LSA time and the number of warnings change with the percentage of the different types of statements. As explained previously, if-else conditions and while loops have minimal impact on the analysis, so our focus was to assess the impact of variable declarations, assignments, and function calls on the analysis. We conducted three separate evaluations, each involving varying the percentage of one of these statement types while keeping the others constant. Note that, to maintain consistent conditions across the tests, we sought to preserve the ratio between all constant parameters. For instance, when studying the impact of variable declarations and assignments, we ensured that the ratio between function calls and if-else conditions remained unchanged. Likewise, while varying the number of function calls, we aimed to keep constant the ratio between standalone function calls and those in assignments.

The plot in Figure 7.1 displays how the average LSA execution time varies with the percentage of variable declarations, ranging from 0% to 100%. Similarly, the plots in Figures 7.2 and 7.3 illustrate how the average LSA execution time changes depending on the percentage of assignments and function calls, respectively, within the range of 10% to 90%. The number of lines was kept constant at 50,000, and, for each configuration, we ran 10 different randomly generated files. Each of the three plots shows, for each configuration, the compile times with and without LSA, as well as the average number of warnings.

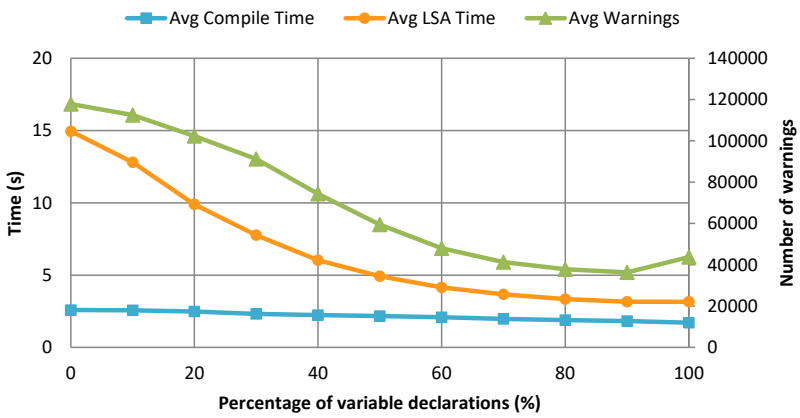

Figure 7.1: Compile time, LSA time, and number of warnings generated per percentage of vars.

In the plot in Figure 7.1, we observe that both the LSA time and the number of warnings steadily decrease as the percentage of variable declarations increases. We conclude that the number of variable declarations has minimal impact on the performance of the analysis.

Note that, even the instances where the main function comprises only variable declarations must be analyzed and still produce more than 40,000 warnings. This is because each variable declaration may also be initialized and therefore they also create dependencies between variables.

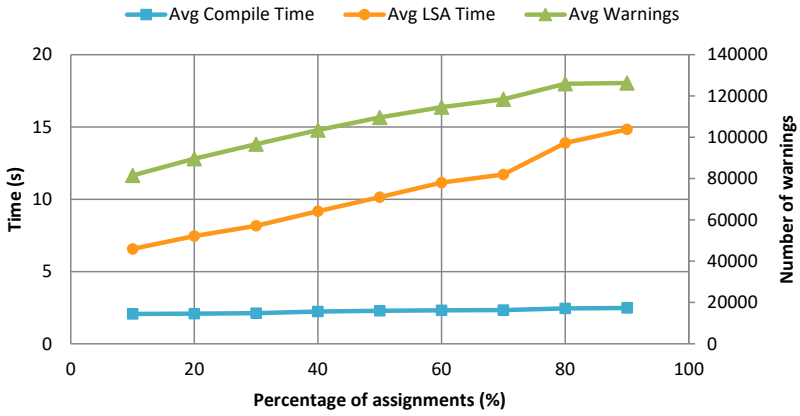

Figure 7.2: Compile time, LSA time, and number of warnings generated per percentage of assignments.

The plots in Figures 7.2 and 7.3 show that the LSA time and the number of warnings increase almost linearly with the percentage of assignments and function calls, respectively. This indicates that assignments and function calls, particularly those involving parameters with multiple levels of indirection, are the constructs that require the most computational effort to validate and also tend to generate the highest percentage of warnings per line of code. Therefore, the bottleneck of the analysis lies in the validation step of assignments and function calls.

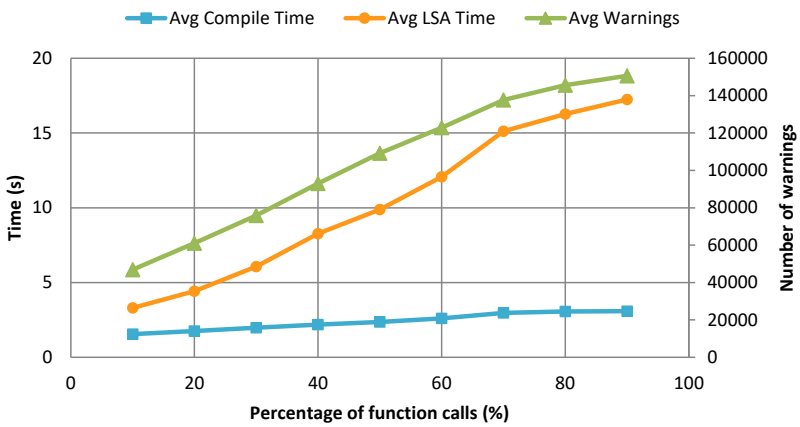

Figure 7.3: Compile time, LSA time, and number of warnings generated per percentage of function calls.

Furthermore, it seems that the higher the number of warnings generated, the longer the analysis seems to take. In light of this, the bottleneck of LSA may specifically be the process of printing the warning messages, during the validation step. Typically, Clang does not optimize this process because, when an error or warning is generated, the program is likely to be rejected. Therefore, it is also important to evaluate the LSA time on programs that do not produce any warnings, which is studied in Section 7.2.

The previously mentioned decrease in Figure 7.1 can therefore be explained by the fact that, as the percentage of variable declarations rises, the number of assignments and function calls decreases, leading to reductions in both the LSA time and the number of warnings. The compilation time remains approximately constant in all of these three plots.

Proceeding from here, we studied how the number of lines of code impacts the LSA time, keeping the percentages of each statement type constant. The plot featured in Figure 7.4 was produced by fixing the percentage of variable declarations on 20% and 60%, while varying the number of lines of the main function across a range from 1,000 to 100,000. All other parameters were set to their default values. Once again, for each number of lines of code, we conducted 10 different tests and calculated their average times.

The LSA time demonstrates an approximately linear increase as the number of lines of code rises, regardless of whether the percentage of variables declarations is set at 20% or 60%. However, the slow-down is less pronounced when 60% of the lines of code are variable declarations, as expected from the conclusions drawn from Figures 7.1, 7.2 and 7.3. The plot in Figure 7.5 helps visualize how the number of warnings scales with the number of lines of code, for both percentages of variable declarations.

While the plot in Figure 7.5 may not offer valuable insights into LSA, it does demonstrate that the number of invalid lifetime associations generated by the script grows linearly with the number of lines. Furthermore, it evidences that the number of warnings increases at a reduced rate when the percentage of variable declarations is higher.

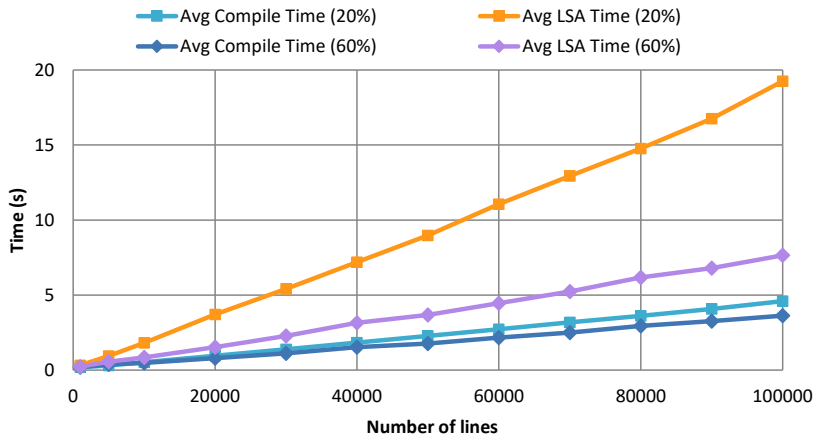

Figure 7.4: Compile time and LSA time per number of lines of code.

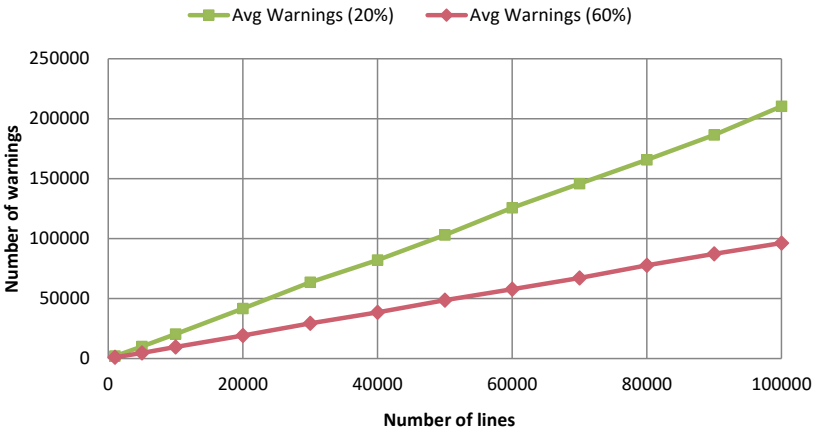

Figure 7.5: Number of warnings generated per number of lines of code.

The final plot, depicted in Figure 7.6, illustrates the variations in LSA time and in the number of warnings generated based on the number of possible lifetime annotations, which range from 1 to 10. The number of lines of code was fixed at 50,000, with the percentages of all statement types set to their default values.

The conclusion we take from the last plot is that the execution time of LSA and the number of warnings are essentially independent of the number of possible lifetime annotations. Even when the only possible lifetime annotations are `$a`, `$static`, and no annotation, LSA generates approximately the same number of warnings.

It is important to note that the code produced by the script places a significant load on LSA because the instances analyzed in this section are essentially the worst-case scenario. The generated code features very long functions with lots of dependencies between variables and the proportion of constructs that LSA has to validate is substantially higher than in practice. Moreover, since the analysis is

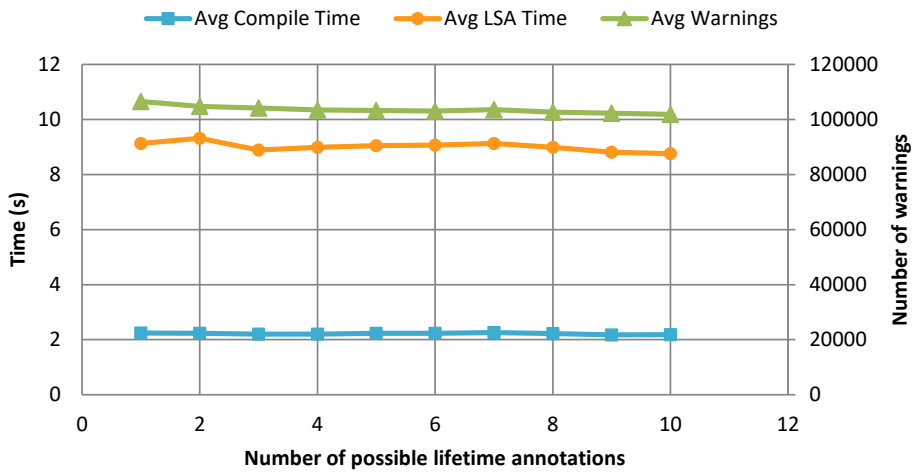

Figure 7.6: Compile time, LSA time, and number of warnings generated per number of possible lifetime annotations.

intra-procedural, the analysis of long functions requires extensive propagation and verification, whereas analyzing several shorter functions is significantly faster. Therefore, the slowdown observed in the plots above will not be as severe when LSA is applied to real-world codebases.

7.2 Case studies

Crubit, the C++ lifetime annotations inference tool, currently cannot perform lifetime inference on entire code files, being limited to unit tests. Without an automated inference tool, we can only execute LSA on codebases if we manually annotate them beforehand. This process is very time-consuming and cumbersome. As a result, we opted to run LSA only on two benchmarks: Bzip2 and SQLite3.

The chosen benchmarks are both written in C, which has a relatively simpler grammar when compared to C++. Therefore, the analysis of C programs is easier and requires a simpler implementation of LSA. Additionally, for the same reason, annotating unfamiliar codebases in C is also less challenging.

It is important to note that LSA supports all C language constructs, except for those related with the heap. As discussed in Section 8.6, we decided not to perform any lifetime analysis on heap allocation. However, both Bzip2 and SQLite3 contain dynamically allocated variables. We handle the heap by attributing the lifetime `$static` to the return value of the `malloc` function. Although this approach may not be entirely correct, it serves the practical purpose of enabling effective analysis of these benchmarks.

In Table 7.1, we present a comparison between the Bzip2 and SQLite3 benchmarks.[4] The compile and LSA times were obtained by calculating the average of 10 executions.[5] In the following subsections, we explore these two benchmarks in detail.

[4] Information retrieved from `https://openhub.net/`
[5] The benchmarks were compiled with the `-O0` flag.

Table 7.1: Comparison between the Bzip2 and SQLite3 benchmarks.

Characteristics	Bzip2	SQLite3
Lines of Code	9.5K	248K
Number of Warnings	23	1310
Number of Annotations	23	1172
Annotation Process Time	2h	30h
Number of Annotations per 10K Lines of Code	2.42	4.73
Annotation Process Time for 10K Lines of Code	12 min	7 min
Compile Time	0.29s	2.40s
Compile + LSA time without Annotations	0.30s	2.63s
Compile + LSA time with Annotations	0.29s	2.57s
Slowdown without Annotations	3.1%	9.6%
Slowdown with Annotations	2.0%	6.8%

7.2.1 Bzip2

Bzip2 is an open-source data compression tool and it was selected as the first benchmark for several reasons. Firstly, it allowed us to apply LSA to a real-world program. Despite this, Bzip2 is relatively short and simple, which was essential since it was the first application of LSA to a non-test program and thus we needed to introduce some modifications on LSA to prevent it from crashing during the analysis.

Without any lifetime annotations, LSA generated only 23 warnings, and thus the process of manually annotating Bzip2 was relatively easy, taking no more than 2 hours. Although some new warnings appeared during the annotation process, in the end, only 23 lifetime annotations were required to eliminate all warnings. The only lifetime annotations used were `$a` (17 annotations) and `$static` (6 annotations).

The slowdown of running the LSA on Bzip2 is insignificant, as well as the difference between analyzing it with and without lifetime annotations, as observed in Table 7.1. This is because most functions are relatively small, which means that there are few dependencies between variables, as explained in Section 7.1. Additionally, most pointers have only one level of indirection, which makes dependencies between them easy to process, as well as making function call arguments easier to check. In fact, LSA rarely has to verify dependencies between arguments in function calls and assignments between pointers with more than one level of indirection, in Bzip2. In other words, to analyze Bzip2, the simpler analysis from Section 5.2 is almost sufficient.

7.2.2 SQLite3

SQLite3, a small and high-performance database engine, is a fundamental component of most mobile phones and computers, with over one trillion databases in use. Its implementation comprises more than 100 files, but there is a version, known as SQLite amalgamation,[6] that combines them into a single large file, with over 200k lines of code. Having the whole code in a single file simplifies the analysis process,

[6] https://www.sqlite.org/amalgamation.html

which makes SQLite3 an excellent benchmark for LSA.

Nevertheless, SQLite3 remains a relatively short benchmark compared to other codebases such as the Chromium project, which comprises over 28 million lines of code. Its implementation is significantly more complex than Bzip2's, featuring substantially larger functions and many more dependencies among variables. Additionally, SQLite3 also contains a much larger number of complex constructs such as pointers with multiple levels of indirection, structs, and function pointers. Therefore, SQLite3 allows us to evaluate the performance of LSA on a more complex codebase.

SQLite3's size and complexity explain why it requires many more lifetime annotations than Bzip2. As illustrated in Table 7.1, SQLite3 requires approximately twice as many lifetimes per 10K lines of code compared to Bzip2. The process of removing the warnings in SQLite3 by manually adding lifetime annotations to the code was not as seamless as in Bzip2 and took approximately 30 hours. The code is much more complex and there are several functions with thousands of lines of code. In fact, we were not able to remove all warnings, ending up with 24 false positives. All of these are related to the fact that our analysis is field-insensitive and thus not precise enough.

The following code is a simplified version of the function `sqlite3VdbeExpandSql`, from SQLite3's implementation. It serves as an example of a false positive that LSA generates due to its field-insensitive analysis.

```
1  void sqlite3StrAccumInit(
2      StrAccum *p, sqlite3 *db, char *zBase, int n, int mx);
3  void sqlite3_str_appendchar(StrAccum *p, int N, char c);
4  char *$a sqlite3StrAccumFinish(StrAccum *$a);
5
6  char *$a sqlite3VdbeExpandSql(
7      Vdbe *$a p,          /* The prepared statement being evaluated */
8      const char *zRawSql /* Raw text of the SQL statement */
9  ) {
10   int n;
11   Mem *pVar;
12   StrAccum out;      /* Accumulate the output here */
13
14   sqlite3StrAccumInit(&out, 0, 0, 0, p->db->aLimit[SQLITE_LIMIT_LENGTH]);
15   sqlite3_str_append(&out, zRawSql, n);
16   if (pVar->flags & MEM_Null) {
17     sqlite3_str_append(&out, "NULL", 4);
18   } else if (pVar->flags & (MEM_Int | MEM_IntReal)) {
19     sqlite3_str_appendf(&out, "%lld", pVar->u.i);
20   }
21   // (...)
22     sqlite3_str_append(&out, "'", 1);
23   }
24   return sqlite3StrAccumFinish(&out); // warning
25 }
```

Even though this function is valid, LSA generates the following warning on the return statement:

```
sqlite3.c:90754:10: warning: function should return data with lifetime '*$a'
                                but it is returning data with lifetime '*$local'
    return sqlite3StrAccumFinish(&out);
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
sqlite3.c:90648:12: note: declared with lifetime '*$local' here
    StrAccum out;            /* Accumulate the output here */
    ~~~~~~~~~~~~
```

The struct `StrAccum` contains a pointer `db` to a database (which may be `NULL`), a string `char *zText`, and several fields of type `int`. Inside the function `sqlite3VdbeExpandSql`, the code creates `out`, a local instance of the struct `StrAccum`.

The function `sqlite3StrAccumInit` takes care of the initialization of the struct. In this case, it sets both `db` and `zText` to `NULL`. After the call to `sqlite3StrAccumInit`, there are multiple calls to the function `sqlite3_str_append`. This function appends text to the `zText` field of the `out`, increasing the size of the memory allocation for the object, if necessary. Finally, the function `sqlite3StrAccumFinish` takes the address of `out` and returns the field `zText` of the struct, which is a pointer. This is valid because, even though `out` is local, the field `zText` is allocated on the heap and thus can be correctly returned from the function.

However, from the point of view of the lifetime analysis, the lifetime of `&out` is local because it is referencing a local struct. Since the analysis is field-insensitive, all fields of `out`, including `zText`, have the same lifetime, which is `$local`. The return value of `sqlite3StrAccumFinish` must have the same lifetime as its only parameter. Alternatively, it could be `$static`, but then the parameter would have to be `$static` as well, which is not possible for the same reason. Consequently, if the lifetime of `&out` is `$local` and the field that is being returned by the function must have the same lifetime, LSA generates a warning stating that the return value is `$local`, which is invalid.

Nonetheless, the pointer returned from is actually valid, which means that we are dealing with a false positive. This could be fixed by changing our analysis to be field-sensitive, gaining precision but also complexity and slowdown.

As mentioned previously, we assume that the lifetime of a heap allocated pointer is `$static`. However, even if the analysis correctly handled the heap, it would still produce a warning on the above code, due to its field-insensitive nature.

All false positives generated by LSA on SQLite3 are similar to this one. They all involve a specific struct field that has lifetime `$static` but, since the analysis is field-insensitive, it is not possible to assign `$static` to a field independently of the others. Instead, its lifetime must match the struct's.

Below is another interesting example to discuss in SQLite3. The `assert` on the fourth line of function `pcacheMergeDirtyList` guarantees that `result.pDirty` will be initialized inside the loop and will not be a dangling pointer. However, since our analysis is flow-insensitive, we cannot take this context into account and thus the analysis perceives that `result.pDirty` may be uninitialized when it is returned.

```
static PgHdr *$a pcacheMergeDirtyList(PgHdr *$a pA, PgHdr *$a pB){
  PgHdr result, *pTail;
  pTail = &result;
  assert( pA!=0 && pB!=0 );
  for(;;){
    if( pA->pgno < pB->pgno ){
      pTail->pDirty = pA;
      pTail = pA;
      pA = pA->pDirty;
      if( pA==0 ){
        pTail->pDirty = pB;
        break;
      }
    }else{
      pTail->pDirty = pB;
      pTail = pB;
      pB = pB->pDirty;
      if( pB==0 ){
        pTail->pDirty = pA;
        break;
      }
    }
  }
  return result.pDirty;
}
```

There are two possible approaches here. LSA could generate a warning, which would prove a false positive in this case. Furthermore, there are other checkers in Clang whose job is to report this kind of error. The second approach is to accept this and not generate a warning, meaning that LSA does not report a false positive in this case, but may miss similar errors in other instances.

Contrary to Bzip2, annotating SQLite3 required using 4 lifetime annotations: `$a` (1052 annotations), `$b` (16 annotations), `$c` (4 annotations), and `$static` (100 annotations). The `$static` lifetime was necessary for global variables, function pointers, and heap allocation. The `$b` and `$c` were used in cases where we needed to establish a lifetime relationship between two variables, independent of other variables already annotated with `$a`. Using different lifetimes allows us to establish different relationships without having to create restrictions between variables that are not associated. These lifetimes are also useful when we need the inner level of indirection of a parameter to be annotated so that the lifetime `$dead` is not attributed to the respective argument. This case is illustrated in the following example from SQLite3:

```
int sqlite3FkLocateIndex(
  Parse *$b pParse,       /* Parse context to store any error in */
  Table *$a pParent,      /* Parent table of FK constraint pFKey */
  FKey *pFKey,            /* Foreign key to find index for */
  Index *$a *ppIdx,       /* OUT: Unique index on parent table */
  int *$b *paiCol         /* OUT: Map of index columns in pFKey */
);
```

In function `sqlite3FkLocateIndex`, both `ppIdx` and `paiCol` serve as return values of the function. Without lifetime annotations, their corresponding arguments will be attributed lifetime `$dead`, and consequently, they will not be readable in the callee. To avoid this, both of them require a lifetime annotation in the inner indirection. Since they are not related, we add `$a` to one of the parameters and `$b` to the other.

Finally, the most used annotation is `$a`, just as in Bzip2. This is because the majority of situations where lifetime annotations are required can be solved with lifetime `$a` alone. There are also many annotations `$a` used in situations equivalent to the one below.

```
SQLITE_PRIVATE const char *$a sqlite3BtreeGetFilename(Btree *$a);
```

The function `sqlite3BtreeGetFilename` returns a pointer, which may either have lifetime `$static` if it is global or heap allocated, or the same lifetime as one of the parameters. In most cases, we are dealing with the second alternative, so we must annotate both the return value and the parameter with the same lifetime, usually `$a`. Note that, in Rust, this function would not require lifetimes annotations because of lifetime elision. As discussed in Section 8.2, LSA would greatly benefit from lifetime elision, significantly decreasing the number of lifetime annotations required and the developers' effort in adding them to the code. Lifetime elision would also decrease by a fair amount the number of warnings generated when LSA analyzes SQLite3 with no lifetime annotations.

In the Table 7.1, we can observe that the slowdown of analyzing SQLite3 is minimal, considering that the combined compile and LSA time is less than 10% higher than the compile time alone. Once the code is correctly annotated, removing most of the 1310 warnings, the analysis time becomes faster, although these values are too small to draw any meaningful conclusion.

8

Discussion

Contents

In this chapter, we discuss implementation choices and limitations of LSA, as well as explore potential future work.

8.1 Subtyping conditions

The concept of subtyping or outlive conditions in Rust was introduced in Section 2.2. Recall that in Rust we can specify that lifetime `'a` outlives lifetime `'b` by using the `where` clause, as illustrated in the function below.

```rust
fn simple_function<'a, 'b>(x: &'a i32) where 'a : 'b {
    let p : &'b i32 = x;
}
```

In the majority of cases, these constraints are not crucial because using the same lifetime for all variables is sufficient. For instance, the behavior of the above function would remain unchanged if we had annotated `p` with lifetime `'a` instead. However, in specific situations, it is necessary to explicitly indicate that two variables have different lifetimes and establish an outliving relationship between them.

As explained in Section 2.2, subtyping conditions are not implemented in the C/C++ lifetime annotations and consequently, our C/C++ lifetime analysis does not have access to such information. Nevertheless, LSA is designed in a way that makes it relatively easy to incorporate subtyping conditions into the analysis in the future, should the necessary syntax be added to the C/C++ lifetime annotations implementation.

8.2 Lifetime elision

In Section 2.2, we introduced the concept of lifetime elision. In Rust, the majority of function declarations do not require explicit lifetime annotations, since they are predictable by a set of rules.[1] For instance, if there is exactly one reference in the function parameters and the function returns a reference, then these two references share the same lifetime. Function signatures where this rule applies do not require explicit annotations.

Lifetime elision in C/C++ lifetime analysis would be very useful, simplifying the process of annotating existing C/C++ codebases. Without lifetime elision, the LSA should issue a warning for every single parameter and return value in every single function that lacks lifetime annotations. This approach would not be the most developer-friendly or efficient.

While we could have implemented Rust-like elision rules, it is important to note that lifetime elision is not a central aspect of C/C++ lifetime analysis, which is the primary focus of this thesis. Due to time constraints, we opted for the adoption of a simpler approach, allowing us to allocate more time to the analysis itself. Therefore, we leave lifetime elision as potential future work.

In our approach, function parameters with no lifetime annotation are assigned the `$local` lifetime by default. Most times, the lifetime `$local` is enough for function parameters since it allows them to be freely used within the entire function body, but they cannot be returned. This means that all parameters that are not related to the return value of the function can have lifetime `$local`. On the other hand, when the return value depends on a parameter, the developer is required to introduce the appropriate lifetime annotations.

[1] https://doc.rust-lang.org/nomicon/lifetime-elision.html

The return value in a function can also have no lifetime annotation, in which case it defaults to the lifetime `$local` as well. This means that any lifetime could be returned by that function, which is not correct since `$local` lifetimes should never be accessed outside of the function scope. To address this, inside the current function, we check if the value being returned has lifetime `$local` and if so, we issue a warning. This approach allows any value annotated with a lifetime that outlives `$local` to be returned, without requiring an explicit lifetime annotation in the return value. Note that, apart from global variables and variables stored in the heap, all pointers with lifetimes different from `$local` must depend on the function parameters. Thus, by allowing any variable that outlives `$local` to be returned, it is guaranteed that the returned value will be valid in the callee. Then, in the callee, the value returned by the function is assigned the lifetime `$local`. As a result, this value may be freely used within the callee function's scope, but it cannot be returned by the callee itself. To fix this, the developer can add an explicit lifetime annotation to the return value of the current function, which must depend on the parameters.

In short, the reason for implementing it in this way is to simplify the process of manually annotating existing C/C++ codebases and of removing all generated warnings. Having to add a lifetime to every single function parameter and return value would be a cumbersome task. This approach offers a simpler alternative to the implementation of lifetime elision, while still improving the usability of C/C++ lifetime annotations. Note that this solution is also safe since it starts with the most conservative assumption. By defaulting to the shortest lifetime possible, it ensures that the code does not access invalid pointers, and, in case a longer lifetime is needed, the developer can then manually add the required lifetime annotation.

8.3 Global variables

Global variables have a global scope, allowing them to be accessed and modified by any function within a program. For this reason, global variables of pointer type must be annotated with `$static`, just as in Rust. This constraint ensures that we can only assign to global variables values that also have lifetime `$static`, thus eliminating the possibility of dangling references.

Recall the fundamental rule that a pointer or reference cannot live longer than the value that it points to. When we apply this rule to `$static` lifetimes, we can foresee how the use of `$static` both for global variables and function pointers may lead to an unreasonable propagation of `$static` lifetime annotations. Whenever we assign a variable `x` to another variable `y` with lifetime `$static`, variable `x` must also be `$static`. Otherwise, LSA must report a warning due to an attempt to assign to `$static y` a variable with a shorter lifetime. The propagation of `$static` also happens "backwards": if a function parameter needs to have lifetime `$static` for some reason, then, whenever that function is called, the corresponding argument must have lifetime `$static` as well. This explains why so many `$static` lifetime annotations were required in the analysis of SQLite3.

To prevent the unreasonable propagation of `$static` lifetimes, developers should restrict the use of global variables, although this may not be plausible when annotating pre-existing codebases.

8.4 Flow-insensitive analysis

We assume a single state per function in our C/C++ lifetime analysis. For this reason, variables have the same lifetime throughout the whole function. This assumption introduces precision loss in the analysis when compared with a flow-sensitive alternative. It overlooks situations where a variable may have a shorter/longer lifetime during specific segments of the function but we always assume it has the shortest lifetime possible. However, as explained in Section 5.2, this approach aligns with Rust's lifetime analysis, and, on top of this, it also performs better in terms of compile time and memory usage.

The following example is the C++ equivalent of the function `f4` from Section 5.1. It illustrates an instance where a false positive is generated due to the flow-insensitive nature of the analysis.

```
1 int *$a fn(int *$a x, int *$b y, bool b) {
2     int *p = x;
3     if (b) return p;
4     p = y;
5     return p;
6 }
```

The third line of this example triggers a warning indicating that `p` may have lifetime `$b` when the function should return `$a`. In fact, when the program reaches this point, `p` cannot have lifetime `$b`. However, since the analysis is flow-insensitive, `p` has the same lifetime throughout the entire function body, hence the false positive.

8.5 Field-insensitive analysis

Contrary to Rust's, our analysis is field-insensitive, and consequently all fields within a struct have the same lifetime as the struct itself. This extends to arrays as well.

The largest drawback of field-insensitivity we came across is in storing global and function pointers inside a local struct, which was discussed in Section 7.2.2. Our analysis defines that these two kinds of pointers have lifetime `$static` by default. However, when stored inside a local struct, their lifetime cannot be `$static` due to the field-insensitive nature of the analysis. This will occasionally generate false positives, that cannot be removed through inserting lifetime annotations in the code.

8.6 Heap

In general, we assume that references and pointers either point to memory allocated on the stack or to other references or pointers. In the first case, their lifetime is `$local` because the data they point to exists only during the function's scope, and therefore the references/pointers to it cannot outlive the function. In the second case, they can have an arbitrary lifetime that may outlive the function. Nonetheless, their lifetime is related to the lifetime of the reference/pointer they were assigned. In both cases, the lifetime annotations and the analysis we developed are enough to handle them.

However, data allocated in the heap is different: it allows creating a pointer unrelated to any other pointer/reference. This is a problem because, as explained in Section 2.2, lifetimes by themselves hold no real information. They only have value when used to relate to other pointers/references. Thus, we either declare the heap-allocated pointer with the same lifetime as some other pointer (which may not reflect what we want) or, as an alternative, we declare that pointer with lifetime `$static`.

The data allocated on the heap can be freed at any time. Specifically, it can be allocated in function `f1` and then deallocated only in function `f2`. Once freed, the corresponding pointer cannot be used anymore, which means that its lifetime should not be `$static` but instead should be something that represents that the pointer is alive from the `malloc` function until the `free` function. This is not possible using the current implementation of lifetime annotations in C/C++.

On top of that, the analysis is intra-procedural, which eliminates the possibility of handling pointers created in one function and freed in another function. Additionally, since the analysis is also flow-insensitive and there is one single state per function, even inside the same function body we cannot possibly describe that the pointer is accessible only from the `malloc` instruction until the corresponding `free`.

This problem with the heap is not limited to manual memory management. Containers, such as `std::vector`, also allocate storage dynamically and may reallocate that storage during their lifetime. As a result, the lifetime of `std::vector::data()`,[2] for instance, is shorter than that of the vector itself.

To support heap allocations correctly, the analysis would have to be flow-sensitive instead and we would need to keep information about which pointers are allocated/alive at each program point. With the current analysis, we found no effective way to support this reasoning. Moreover, the C++ core guidelines strongly discourage the use `malloc()` and `free()`, favoring other constructs such as `std::unique_ptr`. On top of this, Rust's lifetime annotations do not cover dynamic allocations. In fact, all manual dynamic allocations in Rust require the use of raw pointers, inside `unsafe` blocks. For these reasons, we made the decision not to perform any lifetime analysis on dynamic allocations.

As explained in Section 7.2, to manually annotate the Bzip2 and SQLite3 benchmarks, we assumed that the result from a `malloc` is a pointer with lifetime `$static`, ignoring the calls to `free`. This as-

[2] The method `std::vector::data()` returns a pointer to the array used internally by the vector.

sumption is not entirely correct since the pointers are not actually accessible throughout the whole code. Nonetheless, this assumption serves a practical purpose by allowing us to analyze and add lifetime annotations to existing C/C++ codebases while minimizing the number of necessary changes to the code. It is important that we can annotate existing codebases without having to rewrite most of the code.

A possible solution to the dynamic allocations problem, while maintaining the analysis flow-insensitive, is to introduce a new lifetime `$heap`, which describes the lifetime of a dynamically allocated pointer. Since the analysis remains flow-insensitive, this approach does not cover the call to `free()`. For this reason, similarly to `$static`, a value annotated with `$heap` also lives for the entire duration of the program. The lifetime `$heap` is weaker than `$static` but outlives all other lifetimes. This enables the analysis to distinguish between dynamically allocated pointers and global variables.

It is important to note that this is a potential solution, but it was not included in the analysis and has not been carefully studied. For this reason, we cannot guarantee how this change would impact the analysis of dynamic allocations.

However, the introduction of a new lifetime `$heap` is a "permissive" approach because it assumes that the allocated pointer remains valid for the entire duration of the program. A more conservative alternative is to make no assumptions about the lifetime of dynamically allocated memory, aligning with Rust's approach.[3]

An alternative method to address the heap problem in C++ would be to either return objects by value or to use the smart pointer `std::unique_str`, thus removing the need for manually allocating data on the heap. In C++, returning an object by value results in a `move` operation rather than creating a copy, which provides an efficient transfer of the returned object. The `std::unique_ptr` is an object that wraps another object, managing it through a pointer. These smart pointers automatically free the object's memory when it goes out of scope, relieving the developer of the responsibility of manual memory management. Consequently, the lifetime of a pointer held by a `std::unique_ptr` is the scope of that `std::unique_ptr`. The exception is `std::unique_ptr::reset()`, which deletes the currently owned object when it is called.

It is important to note that neither of these two approaches is applicable to programs written in C. Moreover, they both require significant changes to pre-existing code. Therefore, while these are interesting solutions when writing C++ code, it is not plausible to refactor entire existent codebases, even as small as Bzip2 or SQLite3, to make them aligned with these solutions.

[3] https://tinyurl.com/y52ka5sp

8.7 Scopes

Our C/C++ lifetime analysis, contrary to Rust's, does not implement scopes. Consider again the first example from Section 2.2, used to introduce the concept of lifetimes in Rust.

```
1  let r;                // Introduce reference: r
2  {
3      let i = 1;        // Introduce scoped value: i
4      r = &i;           // Store reference of i in r
5  }                     // i goes out of scope and is dropped.
6
7  println!("{}", r);   // r still refers to i, compile-time error
```

Rust analysis does not accept this example because the scope where `i` was created ends before the scope where `r` belongs. However, since our analysis does not cover scopes, an equivalent example in C or C++ is accepted.

```
1  void f1() {
2      int i = 1;
3      int *p = &i;
4
5      {
6          int j = 2;
7          p = &j;
8      }
9
10     i = *p; // should generate a warning
11 }
```

In LSA's implementation, the lifetime of `&j` is equal to the lifetime of `&i`, since they are both `$local`. However, in fact, `j` is deleted on line 7, having a smaller scope than `i`, which exists until the end of the function. Consequently, in line 8, `p` points to a value that is no longer available, making the assignment invalid. This error is not captured by our analysis because LSA does not cover scopes.

To fix this, a possible approach would be to create a different lifetime `$local` for each scope. LSA would know, for each line, which `$local` lifetimes are valid and how they relate to each other. For instance, in the example above, the lifetime of `&i` could be `$local1` and the lifetime of `&j`, or any variables defined inside that scope, would be `$local2`. The lifetime `$local2` ends in line 7, at the end of the inner scope, whereas lifetime `$local1` is valid until the end of the function body. Note that both lifetimes are still local, and thus neither of them can be returned from the function. Then, the lifetime of `$p` would be the intersection between lifetimes `$local1` and `$local2`, which is `$local2`. With this information, LSA could generate a warning in the assignment in line 8.

Introducing scopes in the analysis would allow LSA to identify a greater range of errors, such as the kind above. As another example, LSA would also be capable of generating warnings in function calls like the one illustrated in the following example.

```
1  void f2(int *$a *$b x, int *$a y);
2
3  void f3(int *x) {
4      int **p = &x;    // (int**) p has lifetime $local1
5                       // (int*) p has lifetime $local1
6      {
7          int i = 1;
8          int *q = &i;     // (int*) q has lifetime $local2
9          f2(p, q);        // warning: *p must outlive q
10     }
11 }
```

The implementation of scopes in LSA, together with the use of `std::unique_ptr` in C++ as a replacement to dynamic allocation, presents a potential solution to address the heap challenge discussed in Section 8.6. Each `std::unique_ptr` with no annotation would be assigned a specific `$local` lifetime, determined by the scope in which it was created. This would allow a promising validation of pointer lifetimes, which is not possible with the current approach. By implementing scopes alongside the use of `std::unique_ptr`, we have the potential to not only solve the heap problem but also achieve a somewhat flow-sensitive analysis inside each function. For that reason, this should be considered as the next step in this work.

9

Conclusion

Contents

In this thesis, we developed LSA, a static analysis tool designed to validate Rust-like lifetime annotations in C and C++. It supports all C language constructs, except for those related with the heap. As discussed in Section 8.6, we opted not to support heap allocations because we did not find a simple and effective approach. Despite this, the way the heap is handled allows LSA to successfully analyze codebases at least as complex as SQLite3.

On the other hand, when it comes to C++, our analysis is limited to the features common to C. The C++ programming language is considerably more complex, containing additional elements such as lambda functions, classes, and templates. We opted to restrict our analysis to C constructs, due to time constraints.

LSA's analysis is fast and yields a reasonably low number of false positives. Additionally, it features an effective error system. Developers often struggle to understand C++ compiler error messages, which are known to be lengthy and complex, even for simple syntactic errors, e.g., in templates. To prevent this in LSA, for each violated lifetime rule, the tool generates a single, informative, and helpful warning message, along with no more than 10 notes. These notes provide additional information, thereby simplifying

the correction process.

In conclusion, LSA proves successful and practical in verifying lifetime annotations in C/C++. It represents a significant step towards achieving interoperability with Rust. As a result, it stands as an effective approach to address memory-safety vulnerabilities in C and C++.

9.1 Future work

As discussed in Chapter 8, the two most important upcoming steps are to transition the analysis into a field-sensitive one and to implement scopes.

The introduction of structs in our analysis was motivated by the evaluation of LSA. Since both the Bzip2 and SQLite3 benchmarks make use of structs, their implementation was necessary to accurately analyze them. This integration occurred relatively late in the development process and, due to time constraints, we opted for the simpler, field-insensitive approach. However, the syntax for adding lifetime annotations to structs and classes exists and is similar to Rust's syntax. Therefore, transitioning to a field-sensitive analysis requires some modifications in the analysis, but should be reasonably fast and easy to achieve.

On the other hand, the implementation of scopes is expected to be a less straightforward task. It will require dedicating time to study how scopes are implemented in Rust to correctly understand their role in lifetime analysis. Additionally, it is necessary to carefully reflect on the distinctions between scopes in Rust and C/C++, followed by designing the most efficient way to implement them. Therefore, this is a very promising step to achieve a more accurate analysis, but will probably require considerable effort.

In addition to data pointers, C and C++ also feature function pointers. These increase the complexity of the analysis, since, by using function pointers, programs can call functions that cannot be statically resolved to a definition. LSA should be extended to support function pointers and update the analysis state based on indirect function calls.

Finally, implementing lifetime elision would also be a valuable addition to LSA, reducing the burden of annotating C and C++ codebases.

Bibliography

[1] A. Miné, "Tutorial on static inference of numeric invariants by abstract interpretation," Found. Trends Program. Lang., 2017.

[2] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," in BlueHat IL, 2019. [Online]. Available: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf

[3] J. V. Stoep, "Memory safe languages in android 13," in Google Security Blog, 2022. [Online]. Available: https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

[4] J. V. Stoep and C. Zhang, "Queue the hardening enhancements," in Google Security Blog, 2019. [Online]. Available: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html

[5] S.-F. Chen and Y.-S. Wu, "Linux kernel module development with rust," in DSC, 2022.

[6] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating c to safer rust," in OOPSLA, 2021.

[7] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan, "In rust we trust: A transpiler from unsafe c to safer rust," in ICSE, 2022.

[8] N. Shetty, N. Saldanha, and M. Thippeswamy, "Crust: Ac/c++ to rust transpiler using a "nano-parser methodology" to avoid c/c++ safety issues in legacy code," in ERCICA, 2019.

[9] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding memory safety bugs in rust at the ecosystem scale," in SOSP, 2021.

[10] A. Homescu and S. Crane, "Rust 2020: Lessons learned by transpiling c to rust," in Immunant Blog, 2019. [Online]. Available: https://immunant.com/blog/2019/11/rust2020/

[11] M. Brænne, R. Dejanovska, G. Horváth, D. Gribenko, and L. Versari, "Lifetime annotations for C++," in LLVM discourse, 2022. [Online]. Available: https://discourse.llvm.org/t/rfc-lifetime-annotations-for-c/61377

[12] A. Møller and M. I. Schwartzbach, Static program analysis, 2022. [Online]. Available: https://cs.au.dk/~amoeller/spa/spa.pdf

[13] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," Electronic Notes in Theoretical Computer Science, 2008.

[14] D. J. Pearce, P. H. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis of c," ACM Trans. Program. Lang. Syst., 2007.

[15] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in ASE, 2016.

[16] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," Commun. ACM, vol. 53, no. 2, p. 66–75, feb 2010.

[17] S. Mirliaz and D. Pichardie, "A flow-insensitive-complete program representation," in Verification, Model Checking, and Abstract Interpretation: 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16–18, 2022, Proceedings, 2022.

[18] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, 2003.

[19] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams, "The dogged pursuit of bug-free c programs: The frama-c software analysis platform," Commun. ACM, 2021.

[20] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," Electronic Notes in Theoretical Computer Science, 2008.

[21] D. Hutchins, A. Ballman, and D. Sutherland, "C/c++ thread safety analysis," in SCAM, 2014.

[22] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in ATC, 2012. [Online]. Available: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf

[23] E. Stepanov and K. Serebryany, "Memorysanitizer: Fast detector of uninitialized memory use in c++," in CGO, 2015.

[24] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in PLDI, 2007.

[25] D. Dhurjati, S. Kowshik, and V. Adve, “Safecode: Enforcing alias analysis for weakly typed languages,” in PLDI, 2006.

[26] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, “Ccured: Type-safe retrofitting of legacy software,” ACM Trans. Program. Lang. Syst., 2005.

[27] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “Softbound: Highly compatible and complete spatial memory safety for c,” in PLDI, 2009.

[28] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, “Checked c: making c safe by extension,” in SecDev, 2018.

[29] M. Morehouse, M. Phillips, and K. Serebryany, “Tcrowdsourced bug detection in production: Gwp-asan and beyond,” in C++ Russia, 2020. [Online]. Available: https://assets.ctfassets.net/oxjq45e8ilak/Rq21VTBlLGneXCBHsE4L5/f33461251478e598208ac3e3c1f25896/Konstantin_Serebryanyy_Ishchem_bagi_v_prodakshene_vsem_mirom_GWP-ASan_i_chto_dalshe_2020_11_12_05_24_04.pdf

[30] F. Gorter, K. Koning, H. Bos, and C. Giuffrida, “Dangzero: Efficient use-after-free detection via direct page table access,” in CCS, 2022.

[31] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” ACM Trans. Inf. Syst. Secur., 2009.

[32] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, “Enforcing forward-edge control-flow integrity,” in USENIX Security, 2014. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-tice.pdf

[33] M. Castro, M. Costa, and T. Harris, “Securing software by enforcing data-flow integrity,” in OSDI, 2006. [Online]. Available: https://dl.acm.org/doi/10.5555/1298455.1298470

[34] R. Watson, S. Moore, P. Sewell, and P. Neumann, “An introduction to cheri,” Cambridge University, Tech. Rep. 941, 2019. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf

[35] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, “Intel mpx explained: A cross-layer analysis of the intel mpx system stack,” in SIGMETRICS, 2018.

[36] V. Costan and S. Devadas, “Intel sgx explained,” IACR Cryptol. ePrint Arch., 2016. [Online]. Available: https://eprint.iacr.org/2016/086.pdf

[37] C. Lattner and V. Adve, “Llvm: a compilation framework for lifelong program analysis & transformation,” in International Symposium on Code Generation and Optimization, 2004. CGO 2004., 2004.