MASTER

Calling Jasmin from Rust

Introducing an approach to safely interoperate between the Rust and Jasmin programming languages

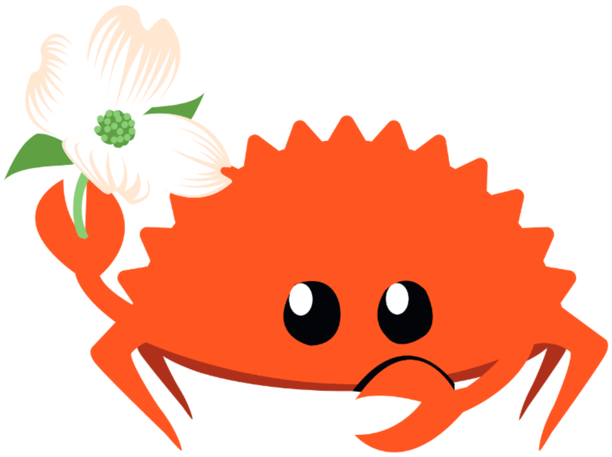
van Drunen, Juriaan

Award date:
2021

Technische Universiteit Eindhoven
University of Technology
Department of Mathematics and Computer Science

Calling Jasmin from Rust

Introducing an approach to safely interoperate between the Rust and Jasmin programming languages



Juriaan van Drunen

Supervisors:
Dr. A. Hülsing (Andreas)
Prof. Dr. P. Schwabe (Peter)
Prof. Dr. M. Barbosa (Manuel)

September, 2021

Abstract

The use and dependence on software systems throughout the lives of people are ever-expanding. As a consequence, the security of these software systems is becoming more important. Advancements in programming languages and tools allow for the writing of more secure software. This thesis introduces an approach to interoperate between the Rust and the Jasmin programming languages safely. Our approach allows programmers to write their non-cryptographic code in Rust, benefiting from Rust's safety and high-level abstractions. And write the cryptographic code in Jasmin, which guarantees memory safety, constant-time code, and predictability. In addition, we provide a tool to make the approach easy for the programmer. Furthermore, we demonstrate our approach by replacing part of the cryptographic code used by Rustls, which is a TLS library written in Rust.

Preface

This work is the outcome of my graduation project for my Master's degree in Information Security Technology at the Eindhoven University of Technology. The thesis introduces an approach to call the Jasmin programming language from the Rust programming language in a safe manner. During my thesis, I learned a great deal about the low-level details of both languages, how they interact and how to analyze the safety of this interaction.

Special thanks go to my supervisors, Manuel Barbosa and Peter Schwabe, for the numerous online meetings and pushing me in the right direction. Their feedback and suggestions were invaluable in the writing of this thesis. I would also like to thank Andreas Hülsing and Nicola Zannone for being part of my thesis committee. Finally, I want to thank my girlfriend, family, and friends for their support and providing me with many distractions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The role of technology throughout the life of people and in organizations is ever increasing as more devices are introduced and interconnected. The reliance and dependence on technology systems are also growing. The security and privacy provided by these systems are therefore becoming more important.

Systems are often compromised through mistakes in the software. Memory errors make up a large part of these vulnerabilities. In 2019 Microsoft reported that 70% of their bugs are due to memory safety issues [20], and the Chromium open-source browser project reported the same number in 2020 [22]. Both projects use C++, which does not provide memory safety. Using a memory-safe language (e.g., Go, C#, Java, Swift, Python) is a way to prevent these memory safety issues. However, for performance and interoperability reasons, unsafe languages like C or C++ are preferred.

Rust is a language introduced in 2010, originally developed at Mozilla Research for writing efficient and reliable system programs [19]. Rust is innovative because of the way it manages memory. Most programming languages use one of the following two approaches:

1. Giving the responsibility to the programmer to allocate and free memory. This allows for great performance as the programmer can carefully optimize memory usage. It, however, comes with the risk of the programmer making mistakes and thus introducing memory-related security issues.

2. Having a garbage collector that checks at runtime if memory is no longer in use and can therefore be freed. This prevents the memory mistakes of the previously mentioned approach but comes at the cost of performance as the garbage collector is constantly running in the background.

Rust takes a different approach by leveraging the type system to manage memory. Rust has the concepts of ownership and borrowing embedded into the type system, with a set of rules that are verified at compile time. The ownership and borrowing concepts inform the compiler which variable owns and has access to the memory and when it is no longer needed. With this approach, Rust provides the same safety as languages using a garbage collector while achieving similar speeds to languages that leave the responsibility of memory management to the programmer. In this way, it addresses the shortcomings of the other memory management approaches at the cost of increased compile-time and some more restrictions on what code is accepted by the compiler. One goal of Rust is to make system-level programming safer through the type system while simultaneously making it more accessible by providing high-level abstractions which do not come at additional runtime cost (zero cost abstractions). Although Rust is a fairly new language, big tech corporations have already expressed interest in the language. Recently (February 2021), the Rust foundation was announced, where the founding members (AWS, Huawei, Google, Microsoft, and Mozilla) participate and financially contribute to Rust and its ecosystem.

In providing security and privacy to systems and communication, there is often a need for confidentiality, integrity, authentication, and non-repudiation. Central to all of these concepts is cryptography. Cryptographic code is also susceptible to memory safety issues. Furthermore, cryptographic code needs to be efficient, functionally correct, and free from secret-leaking side channels. Therefore it is often written in hand-optimized assembly using special programming patterns to achieve constant-time code. However, the hand-optimized assembly and the special programming patterns complicate the verification of the cryptographic code. Jasmin [1] aims to bridge this gap by providing an integrated framework that, along with the Jasmin language, includes tools to reason about the safety of the program and an embedding into the EasyCrypt framework [4] for verifying correctness.

In this thesis, we introduce an approach to interoperate between the Rust and the Jasmin programming languages and discuss the safety of this approach. Being able to utilize Rust and Jasmin together is a useful extension for both languages. The combination makes it possible to write the high-level code in Rust, benefiting from the safety Rust provides and the quickly growing Rust ecosystem. And use the Jasmin framework to write the cryptographic code, achieving maximum efficiency and safe cryptographic code.

The structure of this thesis is as follows. In Chapter 2, the necessary background on Rust and Jasmin is introduced. Chapter 3 discusses the Application Binary Interface (ABI) of both Rust and Jasmin. Chapter 4 describes the approach to interoperate between Rust and Jasmin and presents a tool to automate part of the process. Chapter 5 is a case study of an existing Rust library, where two algorithms written in C and assembly are replaced with Jasmin code, and the safety is discussed. The future work, limitations, and conclusion are provided in Chapter 6.

Chapter 2

Background

2.1 Rust

Rust is a programming language originally developed by Mozilla research for writing efficient and reliable system programs. It provides direct access to the hardware, giving the programmer control over their programs' memory layout and running time. Rust was designed with concurrency and parallelism in mind and provides strong guarantees about memory safety. To achieve this, Rust comes with a strong type system that allows the compiler to statically detect a great number of memory safety issues and data races [19].

Memory safety can be divided into two categories: temporal and spatial. Temporal memory safety ensures that no pointer is dereferenced, which is invalid. Spatial safety ensures that access to a pointer is always within bounds of the object that the pointer is referring to [25]. Rust mainly innovates in the area of temporal memory safety, which it accomplishes through its type system. The type system is not expressive enough to guarantee spatial memory safety. Here the compiler can reason about simple cases to validate that access to a pointer is within bounds but relies on runtime checks for more complex cases.

Rust distinguishes between two kinds of references. Mutable references are unique pointers to a memory location, and immutable references can have multiple pointers to the same memory location. At the heart of Rust's memory safety is the principle of *mutability XOR aliasing*, which guarantees that at any given time, either *one* mutable reference can mutate a memory location or a memory location can be shared immutably many times [13]. This principle helps to rule out many common memory safety issues, as we will discuss below. Enforcing this principle are two key concepts called ownership and borrowing.

2.1.1 Ownership and Borrowing

In Rust, there is always one unique owner of a memory location. This principle is demonstrated in the example program below. In Rust, `Vec`, a vector type, consists of three fixed-size values: pointer, length, and capacity. The pointer points to a buffer of *length* neighboring elements on the heap. The capacity indicates the amount of allocated space for the vector. In the example below, we create the vector containing the numbers 1 to 3 (`vec![1,2,3]`) and assign this to the variable `foo`. On line 2 `foo` is assigned to the variable `bar` and finally, we print the value of the variable `foo`.

```
1       let foo = vec![1,2,3];
2       let bar = foo;
3       println!("{:?}", foo); // Value foo is used after the
4                              // ownership has moved to bar
```

The program fails to compile as the ownership of the vector is moved from `foo` to `bar` on line 2. Having both `foo` and `bar` active simultaneously would violate Rust's unique ownership principle: in Rust, there is always one unique owner of a memory location. When assigning the value `foo` to `bar`, the pointer, length, and capacity are copied (note that the buffer on the heap is not copied). The `foo` variable is then invalidated and can therefore no longer be used. This process is referred to as a `move`.

Rust automatically deallocates values when the owner goes out of scope. How this happens depends on the type. In Rust, types can be divided into two categories. Types that manage a resource that eventually needs to be released or freed when the value goes out of scope and types that can be duplicated by copying the value's bits. To distinguish between the two, Rust uses the concept of traits. Traits are the way for Rust to define shared behavior for types in an abstract way [16]. Types that manage a resource implement the `Drop` trait, specifying how the resource (e.g., memory, file descriptors, locks) the type manages should be cleaned up. Types that can be bitwise copied implement the `Copy` trait. This marker indicates the type can be bitwise copied without needing to take ownership of the original instance. This is not true for the Vec type as it manages a resource on the heap, and a bitwise copy would create two pointers to the same memory location. Types that can be entirely stored on the stack (i.e., the size is known at compile-time) can implement the `Copy` trait. The Drop and Copy traits are exclusive and can not be implemented for the same type: values that manage resources can not safely be bitwise copied without creating two pointers to the same memory location.

When a value goes out of scope, the deconstructor is run, which calls drop for that value if the `Drop` trait is implemented for the type. After this, Rust recursively calls the deconstructor on all the fields of the type. The deconstructor takes ownership of the variable, preventing the usage of the value after it has been deconstructed. The compiler automatically inserts these calls to the deconstructor. The automatic insertion prevents use after free, and the unique ownership prevents a double-free as only the owner of the value can call the deconstructor. Types that implement `Copy` are stored on the stack, and they are automatically deallocated when a function returns, and the stack frame is cleaned up.

```rust
1       let mut foo: i32 = 4;
2       let mut bar: i32 = foo;
3       foo += 1;                   // Increment foo by 1
4       println!("{}", foo);     // Prints 5
5       println!("{}", bar);     // Prints 4
```

The above example declares the values `foo` and `bar` as mutable (with the `mut` keyword). This indicates that the programmer can mutate the values of the variables. This example does not result in a compile-time error because the type `i32` implements the `Copy` trait. In the example above, this would mean that the value 4 gets duplicated on the stack. This, in turn, does not invalidate the previous variable as both variables point to unique locations in memory. Instead of the ownership being *moved*, types that implement `Copy` get duplicated. Many primitive types implement the `Copy` trait (e.g., integers, floats, booleans).

Let us look at a slight alteration of the previous example with vectors again. Here the variables `foo` and `bar` are declared as mutable. On line 3, we mutate the value `foo` by pushing the value 4 to the end of the vector.

```rust
1       let mut foo = vec![1,2,3];
2       let mut bar = foo;
3       foo.push(4);
4       println!("{:?}", bar);
```

Compiling this code results in the following error: *borrow of moved value: 'foo'*. This is because `foo` is used on line 3 after the ownership has moved to `bar`. It would, however, be quite restrictive if only the owner would be allowed to mutate values. It is common that a value is passed to a function mutated in some way and then returned to the caller. `push` is such a function, it takes a collection and a value to be appended to the end of the collection. This would, however, require ownership to be passed back and forth between functions, which is not ideal. To facilitate these common cases, Rust allows for temporary loans of values by the use of references; this is referred to as *borrowing*. Rust has two types of references, mutable references and shared references. First, we describe mutable references.

The function signature of `push` for a vector containing values of type `i32`:

```rust
1       fn push(&mut self, value: i32)
```

The signature takes `&mut` to `self` (`self` refers to the collection push is being called on) and an integer of type `i32`. `&mut` indicates that push takes a mutable reference to the vector, borrowing it for the duration of the call to `push`. It might seem that this contradicts the unique ownership principle. However, the compiler ensures there is at most one unique mutable reference being used at any given time by imposing the following two constraints on mutable references [13]:

1. A reference can only be used during its lifetime.

2. The referent (value being referenced) can not be used as long as active references borrow from the original reference.

To reason about these rules, the compiler has the notion of lifetimes, which specify the span of time in which references are in use. References have additional type information associated with them that indicate the lifetime of the reference. Thus the full type of a mutable reference is `&’a mut T`, where `’a` indicates that this type has lifetime `’a` and `T` indicates the type the reference is referring to.

```
1     let mut foo = vec![1,2,3];
2     let bar: &’a mut Vec<i32> = &mut foo;
3     println!("{:?}", bar);
4     println!("{:?}", foo);
```

In the example above, a mutable reference to vector `foo` is created on line 2. This mutable is annotated with lifetime `’a` and is used on line 3 again. On lines 2 and 3, the reference is thus active, after which it is no longer used. Therefore on line 4, the original referent can be used again without violating the lifetime rules. This program compiles without errors.

```
1     let mut foo = vec![1,2,3];
2     let bar: &’a mut Vec<i32> = &mut foo;
3     bar.push(4); // reference bar is reborrowed
4     println!("{:?}", bar) // reference bar gets used again
```

In the example above, the mutable reference `bar`, borrowed from `foo`, is borrowed again to the push function. This is referred to as a *reborrow*. In the case of reborrowing, two mutable references can exist to one piece of data. This is, however, still in line with the Rust safety rules as the compiler knows that these mutable references alias. The type of mutable reference of `push` is: `&’b mut Vec<i32>`. While the reference with lifetime annotation `’b` lasts, the programmer may use it, but the reference with lifetime annotation `’a` may not. Only after the span of time in which the reference with lifetime annotation `’b` is used ends (after the call to `push`) the mutable reference with lifetime annotation `’a` is allowed to be used again. In this way, the compiler ensures that mutable access to both references stays exclusive.

```
1     let mut foo = vec![1,2,3];
2     let bar: &’a mut i32 = &mut foo[1];
3     foo.push(4);         // original referent is used
4     println!("{:?}", bar); // reference bar gets used again
```

Here `bar` borrows the second element of the vector. The pointer thus points to the location of the `foo` buffer plus the size of an `i32`. Here the original referent gets used before the lifetime of reference `bar` has expired. This violates the lifetime rules and therefore results in a compile-time error. The `push` on line 3 could have altered the location of the `foo` buffer on the heap (due to it being out of space and having to reallocate somewhere else), invalidating the `bar` pointer.

The other reference type is a shared reference. Shared references have the following type information: `&'a T`. Trying to mutate a shared reference will result in a compile-time error as they are immutable. The compiler ensures that shared references adhere to the following constraints [13]:

1. A reference can only be used during its lifetime.
2. The referent (value being referenced) does not get mutated as long as active references borrow from the original reference.

```
1       let mut foo = vec![1,2,3];
2       let bar: &'a Vec<i32> = &foo;
3       let baz: &'b Vec<i32> = &foo;
4       println!("{:?}", baz);
5       println!("{:?}", bar);
6       foo.push(4);
```

On line 2 and 3 `bar` and `baz` borrow from `foo`. The lifetimes of these borrows overlap as `baz` is active on lines 3 and 4 and `bar` on lines 2 through 5. This program compiles successfully as it adheres to the above rules for shared references. Since shared references are immutable, it is not possible to mutate the borrowed value through the references. Shared references can be seen as read-only access to a value.

```
1       let mut foo = vec![1,2,3];
2       let bar: &i32 = &foo[1];
3       let baz: &i32 = &foo[2];
4       println!("{}", baz);
5       foo.push(4);
6       println!("{}", bar);
```

The program above fails to compile as on line 5 the referent `foo` is mutated while there is still a shared reference active (`bar` is used on line 6). The `push` to `foo` could invalidate the pointer of the reference `bar` as it might reallocate the buffer on the heap. This would result in a dangling pointer, but the constraints on shared references prevent this error.

Rust can pass values in two ways: by reference and by value. When passed by value (for non `Copy` types), ownership is moved, and when passed by reference (mutable or shared), the value is borrowed bounded by a lifetime. The difference between mutable and shared references is that multiple shared references can be active simultaneously (i.e., overlapping lifetimes). In contrast, for mutable references, there is always only one active reference at any given time [15].

As explained, Rust prevents double-free and use-after-free through unique ownership of memory locations. The borrowing rules prevent iterator invalidation and data races. Iterator invalidation is prevented as no data structure can be mutated while it is being iterated over. Data races occur when two or more threads access the same memory location, and one of these operations is a write without some form of synchronization [29]. These are prevented as no shared mutable state can exist. Everything discussed up until now is known as safe Rust, which is the default when programming in Rust.

2.1.2 Unsafe Rust

The Rust type system allows for many programs to be written safely. There are, however, data structures that rely on shared mutable state, whose implementation is prevented by the Rust type system [14]. To implement these data structures, Rust allows for the easing of the constraints by using `unsafe` Rust code. Unsafe Rust can be seen as a separate language where some of the restrictions enforced by Rust are relaxed.

One of the restrictions lifted is the ability to dereference raw pointers. In unsafe Rust, the borrow checker and other safety mechanisms are still active, but the borrowing rules do not apply for raw pointers.

```rust
fn main() {
    let mut foo = 1;
    let foo_ptr = &mut foo as *mut i32;
    let res = unsafe { deref(foo_ptr)};
    println!("{}", res) // prints 42
}
unsafe fn deref(foo: *mut i32) -> i32 {
    let bar = foo as *mut i32;
    *foo = 13;
    *bar = 42;
    *foo
}
```

In the example above, an unsafe function `deref` is declared, dereferencing two raw pointers and changing the contents to 13 and 42, respectively. Since the borrowing rules are not enforced for raw pointers, it is allowed (in unsafe Rust) to have multiple mutable pointers to the same memory location. Unsafe Rust with the right restrictions can be safe, but the distinction with Rust is that the compiler can not guarantee this. It is thus the responsibility of the programmer to make sure the written code does not cause any undefined behavior. The `unsafe` keyword makes it explicit where the programmer has relaxed the compiler rules, and when reviewing the code, extra effort should be made to verify the operations in these `unsafe` blocks.

Unsafe Rust can also be used to interface with other languages, like C. Since the Rust compiler can not guarantee the safety of these languages, a call made to another language has to be surrounded by the `unsafe` keyword.

2.1.3 Rust runtime checks

Rust's type system is not able to prevent all memory-safety issues at compile time. Rust relies on runtime checks for preventing division by zero, for checking that memory access is within bounds, and for catching stack overflows. These checks are omitted if the compiler at compile time can validate that these checks are unnecessary (e.g., a value is divided by a constant that is not zero).

Before looking at the runtime checks, it is good to get a high-level overview of how Rust compiles code. The Rust compiler goes through several stages before the eventual binary is generated. First, the Rust syntax is analyzed by the lexer and turned into a token stream. This is then parsed to generate an abstract syntax tree (AST). During this phase, Rust macros are expanded, name resolution is performed, and validation of the AST is done. The AST is converted to a high-level intermediate representation (HIR), which involves desugaring. During translation to HIR, type inference and initial type checking happen. The HIR is then translated to a mid-level intermediate representation (MIR), closely resembling LLVM intermediate representation (LLVM IR). In the MIR phase, borrow checking and optimizations are performed. Rust uses LLVM as a backend for code generation. LLVM provides target-independent optimization and code generation for many CPU architectures. For code generation, LLVM takes as input LLVM IR which closely resembles assembly but comes with added annotations and low-level types [10]. So the MIR is translated to LLVM IR, which performs more optimizations before generating the final binary. Runtime checks for division by zero and bounds checking are introduced at the MIR level. The Rust standard library and LLVM introduce Stack-overflow checks.

Divide-by-zero checks are inserted when the division operator is used. The compiler inserts instructions to compare the divisor to zero. If they are equal, an error is thrown, indicating a divide-by-zero error. When the compiler can determine that the divisor is zero during the MIR compiler phase, the program will fail to compile, indicating the division operation will fail at runtime. If LLVM can determine that the divisor equals zero, the program will compile a program that will print a divide-by-zero error without performing any divide by zero checks. Both MIR and LLVM can remove the runtime check if they know that the divisor will never be equal to zero.

For indexing, the Rust compiler inserts bounds checks. In some cases, during the MIR phase, the compiler can reason that the index is out of bounds, resulting in the program failing to compile with an error indicating that the program will crash at runtime. If LLVM can infer that a program will index out of bounds, the program will compile successfully but will crash by default with an out-of-bounds error. Bound checks are inserted when the compiler lowers index expressions in the MIR compiler phase. For example, consider the Rust function below:

```rust
pub fn increase_by_pos(slice: &mut [u8]) {
    slice[3] = 5;
}
```

The function gets a pointer to a mutable slice of memory and assigns the value 5 to the fourth position in this slice. A slice is a dynamically sized type (the size is not known at compile time) and therefore uses fat pointers. Therefore, a pointer to slices in Rust consists of the memory location of the slice and the number of elements in the slice. When the compiler evaluates this index expression, it inserts a check that verifies that the value used to index the slice is less than the number of elements in the slice. LLVM can remove the out-of-bounds checks when it can reason that the indexing will never be out-of-bounds.

Stack-overflow protection is implemented by relying on guard pages and LLVM stack probes. If the stack overflows and the program tries to read or write to a guard page, the program will segfault. The details of how this is implemented are platform-specific, e.g., Linux automatically sets up a guard page for the main thread of a program, but for additional threads, guard pages are set up by Rust with a call to `libc mprotect`. Stack probes[1] are inserted when a function requires more stack size than the guard page. Stack probes are used to prevent the stack clash vulnerability [23]. Here a stack overflow jumps over the guard page(s) and accesses a mapped memory region below the guard page(s). Stack probes ensure that every page belonging to the stack is accessed, ensuring the guard page is hit if the stack overflows. The details of stack-overflow protection are platform-specific and part of the Rust runtime. When programming using the `#![no_std]` attribute, which only relies on the platform-agnostic core Rust library, used for embedded development, guard pages are not set up by Rust. In this case, the programmer has to either ensure the targeted environment sets up guard pages or take the responsibility to prevent stack overflows.

2.2 Jasmin

Jasmin is a framework for developing cryptographic software which is highly efficient and provides strong safety guarantees [1]. Good cryptographic software must be efficient, functionally correct, and free from any secret leaking through side channels.

Cryptographic software is often hand-optimized in assembly to meet the efficiency requirements using special programming patterns to achieve constant-time code. Constant-time code does not leak secret information (e.g., keys) gathered by measuring the execution time of cryptographic implementations. Both the efficiency requirements and the special programming patterns complicate the verification of the cryptographic code. Jasmin aims to bridge this gap by providing an integrated framework that, along with the Jasmin programming

[1] Currently, stack probing is only supported on x86. Work is being done on implementing this for AArch64

language, provides tools to verify the safety of the program and an embedding into the EasyCrypt [4] framework. EasyCrypt supports proofs of functional correctness, functional equivalence, and the ability to prove constant-time code and crypto reduction proofs with minimal user interaction. Jasmin comes with a certified compiler that guarantees that the assembly program behaves the same as the Jasmin program. This allows safety and correctness reasoning to happen at the Jasmin source code, which greatly simplifies the process.

2.2.1 Jasmin language

The Jasmin language has both high-level structures (e.g., `for` loops), which are easier to verify, and low-level instructions, giving the programmer precise control over the generated code. One of the core principles in Jasmin is predictability. The programmer will be able to envision how the assembly will be generated, which allows for the writing of highly efficient code. Below is a fictive example of cryptographic code written in the Jasmin language. It demonstrates some of the features of the language, which we discuss below.

```
export fn main(reg u64 key, reg u32 state)
{
    reg   u32[16] k; // the full state is in k[0..14] and k15;
    stack u32     k15;
    stack u32     k14;

    k, k15 = init(key, state);
    rotate_left(k, 0, 4, 8, 16);
    rotate_left(k, 2, 6, 10, 16);

    k14 = k[14];
    k[15] = k15;

    rotate_left(k, 1, 5, 9, 16);
    rotate_left(k, 3, 7, 15, 16);

    k15 = k[15];
    k[14] = k14;
}
```

The function `main` is annotated with the `export` attribute. This function is the entry point of the Jasmin program. By default, functions are automatically inlined during compilation[2] and therefore not present in the generated assembly.

In Jasmin, the programmer specifies where data is allocated using the keywords: `inline`, `stack`, `reg` and `global`. `inline` values are resolved at compile-time, `stack` values are stored on the stack, `reg` values are stored in registers,

[2]Since the writing of this section Jasmin has introduced several new features, most notably function calls, global arrays, and the ability to have a register pointer to a stack array. Due to Jasmin introducing function calls, functions are no longer inlined by default.

`global` values are stored in the code segment (requires the value to be known at compile-time). This gives great control over the generated assembly, which allows for achieving maximum efficiency.

When there are more live variables than registers, compilers may perform register spilling. Here the content of some registers is transferred to memory to free up registers for other values. The Jasmin compiler does not automatically perform register spilling. The programmer should thus explicitly handle register spilling. This is demonstrated in the example above, where the variable `k`, which is a collection of 16 registers, and stack variables `k14` and `k15` are defined. On the `x86_64` architecture, 15 registers are free to be used simultaneously. Therefore `k15` is initially stored on the stack. When `k15` is needed, the register holding `k14` is stored on the stack, and `k15` is placed in a register. Not automatically performing register spilling ensures predictability of the generated assembly.

```
1  fn init(reg u64 key, reg u32 state) -> reg u32[16], stack u32
2  {
3    inline int i;
4    reg   u32     k15;
5    reg   u32[16] k;
6    stack u32     s_k15;
7
8    k15 = state[13];
9    s_k15 = k15;
10
11   for i=0 to 2
12   {
13     k[i] = (u32)[key + 4*i];
14   }
15   for i=2 to 14
16   {
17     k[i] = state[i-2];
18   }
19
20   return k, s_k15
21 }
```

Jasmin allows for the reading and writing operations of all word sizes to and from stack arrays. As seen in the above example (line 13), values from the pointer key are read and interpreted as `u32`.

On lines 11 and 15, a `for` loop is used. These high-level structures are normally not used when writing cryptographic code due to the overhead of keeping a counter, but the Jasmin compiler automatically unrolls `for` loops. Using high-level structures makes the code more intuitive and compact to write without a performance penalty; the compiler preserves `while` loops.

```
1 fn rotate(reg u32[16] k, inline int a, b, c, r) -> reg u32[16]
2 {
3   k[a] += k[b];
4   k[c] ^= k[a];
5   _, _, k[c] = #ROL_32(k[c], r);
6   return k
7 }
```

Jasmin can use specific CPU instructions of the architecture it supports (currently only `x86_64`). On line 5, the `ROL` instruction is called, which rotates the bits to the left. CPU flags are handled explicitly in Jasmin. In the above example, the `ROL` instruction might cause the overflow flag (`OF`) and carry flag (`CF`) to be set. The calling of the `ROL` instruction thus returns the result and the flags. By handling flags in this way, the side effects are explicit from the program code. Except for memory-related instructions, all instructions are treated as pure operators, making state modifications explicit. This simplifies the reasoning about side-channel resistance and functional correctness.

To achieve maximum performance, cryptographic code often makes use of vectorized instructions provided by the architecture. Jasmin supports SIMD extensions of the `x86_64` architecture (not shown in the example). The embedding in EasyCrypt allows for proving equivalence between the functional specification and an implementation with vectorized instructions.

2.2.2 Jasmin safety

Jasmin comes with a static analyzer to prove the safety of the program. The static analyzer guarantees the program terminates (when able to), in-bounds array access, valid memory access, absence of division by zero, and that all variables are initialized.

For a program to terminate, it needs to hold that there is a finite set of transitions for any initial state. For proving this, an approach based on ranking functions is used [2]. This approach can infer that the number of transitions between program states will decrease, eventually ending the program. The checking for termination is incomplete, meaning the safety checker cannot prove termination for all programs. When the safety checker is unable to prove termination, it will report the program does not terminate. This means it can not prove termination, and the programmer should ensure the program terminates.

The absence of division by zero, out-of-bounds array accesses, and variable initialization are checked during the Jasmin safety analysis. The safety analysis will return errors if any of these safety conditions are violated. To statically reason about the program safety, the Apron library [12] is used, which is able to represent variables in the polyhedra abstract domain. In the polyhedra domain all possible linear constraints between variables can be encoded [26], allowing to prove the absence of, e.g., division-by-zero and index out-of-bounds.

A Jasmin state consists of the global memory available to all functions and the local environment, only available to the current function. Jasmin requires that all read and write operations happen on valid memory. Valid memory adheres to the specific architecture's constraints (e.g., alignment) and should be initialized. Jasmin has a simple form of dynamic allocation, where Jasmin allocates memory on the stack for function-local variables upon function entry and frees the memory upon function exit. Jasmin can work with pointers; however, Jasmin cannot verify if the memory locations supplied to the entry point are valid. Therefore Jasmin computes an overapproximation of the memory regions accessed as a precondition, which - when satisfied - guarantees safe execution. The programmer has to ensure that these preconditions hold. Compiled programs may end up consuming more stack space than the source program (e.g., due to stack alignment requirements of the target platform). This means that a safe source program under the safety preconditions might still fail due to a lack of stack space.

To prove formal security, functional correctness, and constant-time code, Jasmin provides an embedding in the EasyCrypt framework. The EasyCrypt framework is designed for the verification of cryptographic code. The soundness of the embedding requires the Jasmin program to be safe. As Jasmin programs are statically analyzed to guarantee safety at compile-time, this is always the case. The safety assumption simplifies the verification conditions generated and, therefore, the proof of functional correctness and functional equivalence properties [2]. For constant-time code, it should hold that any two executions with only different secret information (i.e., all public values are the same) should result in the same information leakage [3]. To prove this condition, traces are added to the EasyCrypt embedding of the Jasmin program for branching conditions and memory access. Then it is verified that two given programs with the same public state will produce the same leakage trace. The programmer will need to verify that the variables inferred to be public are indeed public.

Chapter 3

Rust and Jasmin ABI

Rust provides a Foreign Function Interface (FFI) to interact with other languages and the underlying OS. The most common use of the FFI is the interaction with the C language. When calling into C code, the programmer can tell Rust to use the C ABI by using the `extern` keyword:

```rust
extern "C" {
    fn plus_one(foo: i32) -> i32;
}

fn main() {
    unsafe {
        println!("1 + 1 equals {}", plus_one(1));
    }
}
```

This ensures that the function uses the standard C ABI of the platform when the external function `plus_one` is called. As the Rust compiler cannot verify that the external code adheres to the Rust safety rules (i.e., the code is memory safe), calls using the FFI need to be wrapped with the `unsafe` keyword. This is not ideal as the interoperation between Rust and Jasmin should uphold the guarantees of both languages, which implies memory safety.

Therefore another approach is considered where we pack the Jasmin code into a Rust library file, which can be called from a Rust program without relying on the `unsafe` keyword. In this chapter, we examine the Rust and Jasmin ABI.

3.1 Rust Rlib and ABI

By default, an Rlib file, which is a static Rust library with additional metadata, is generated when compiling a Rust library. An Rlib file is a .ar archive on Linux, which groups files as a single file with additional metadata. The archive file contains the object files of the library and a file that describes the metadata of the library. The metadata includes, among others, the Rust compiler version, dependencies, language attributes, code location of all definitions, MIR of inlined functions and generic functions, and type and macro definitions. The Rlib format and metadata format have yet to be fully standardized[1].

When compiling a Rust program that depends on a Rust library, the compiler queries the metadata of the external Rlib to ensure the calling code is correctly typed and to ensure the library functions are called correctly (e.g., the correct number of arguments). The Rust compiler does not verify the safety of the external library, as the safety has already been checked during the compilation of the library. Memory safety of the external library is guaranteed either at compile time or by introducing runtime checks when appropriate. At the end of the compilation process, the Rlib files are passed to the linker, consuming the object files within the Rlib to generate the final executable.

Rust by default uses symbol mangling; this allows for functions of different modules to have the same name without naming conflicts arising during linking. To interoperate with external code, Rust provides the `#[no_mangle]` attribute that prevents the mangling of function names. By preserving the function name during the compilation of the Rlib file, the library function will be called by Rust without mangling the function's name. Rust also provides the `#[export_name]` attribute, which can be used to give a custom name to a function or static value in the final output file [28].

The compiler removes static values that are not used in the Rust code. To preserve a static value, the programmer can add the `#[used]` attribute to the static value. This will prevent the compiler from removing the value from the output file. The linker can still remove items marked with the `#[used]` attribute [28].

To force the compiler to place the contents of a function or static value into a specific section of the object file, the `#[link_section]` attribute can be used. This allows for precise control over the exact location where the final code is placed with the help of the linker [28].

```rust
// Static foo will not be removed from the compiled output file
#[used]
static foo: u32 = 42;

// Function bar preserves the function name (no symbol
// mangling) in the final compiled output file and is
// placed in the section .bar_section in the object file
#[no_mangle]
#[link_section = ".bar_section"]
pub fn bar() {

```

[1]During the research, we used the following compiler versions: rustc 1.51.0 (2fd73fabe 2021-03-23).

```
}

// Function baz has the name baz_function in the
// compiled output file
#[export_name = "baz_function"]
pub fn baz () {

}
```

Rust on `x86_64` uses the calling convention from the `System V AMD64 ABI`. This entails that when passing arguments to a function, the first six arguments that fit in a general-purpose register are passed in registers: RDI, RSI, RDX, RCX, R8, R9. If additional arguments are provided, they are passed via the stack. For passing floating-point numbers or types that do not fit into the general-purpose registers, the vector registers xmm0 up to xmm7 are used.

The size and alignment of a subset of Rust types are shown in Table 3.1.

Type	Size in bytes	Alignment in bytes
Bool	1	1
u8/i8	1	1
u16/i16	2	2
u32/i32	4	4
u64/i64	8	8
u128/i128	16	8
usize/isize	8	8
&/&mut	8	8
const/*mut	8	8
[T; N]	size of T * N	alignment of T
[T]	size of T * N	alignment of T
&[T]/&mut[T]	16	alignment of T

Table 3.1: Rust type size and alignment on x86_64

The size and alignment of most types are straightforward. The `bool` type can only take the values 0x00, to indicate `false`, and 0x01 to indicate `true`. Any other value is undefined behavior. `usize` and `isize` are platform-specific types and are guaranteed to be the same number of bits as the platform's pointer type (`x86_64` in this case). Therefore, pointers and references take the same size and alignment as `usize/isize`. Arrays (`[T; N]`) are laid out in memory sequentially, meaning that given the address of the array, the n'th element can be accessed as follows: $address + n * size\_of(T)$. Slices (`[T]`) have the same layout as arrays. Types of which the size is not known at compile time are known as dynamically sized types. References or pointers to such types have twice the size of normal pointers, also known as fat pointers. In the case of a slice (`&[T]/&mut[T]`), this pointer contains the memory location and the number of elements in the slice.

When encrypting data, there is often an input buffer that contains the data to be encrypted and an output buffer to which the encrypted data is written. Both of these are of unknown size at compile time. In Rust, the idiomatic way of passing a pointer to a contiguous memory region is by using slices. As mentioned, slices are dynamically sized; therefore, the pointer to a slice consists of the memory location and the number of elements in the slice. When passing a slice to a function, this ends up using two registers, one for the memory location and one for the number of elements. When dealing with an input and output buffer, this would thus consume 4 registers. However, when the input buffer and output buffer are of the same length, only one argument to specify the number of elements of both buffers is sufficient, therefore ideally, only three registers would be needed. Furthermore, it may be desirable to have the input and output buffer overlap, such that less memory is needed to perform the crypto routine. The data to be encrypted is read-only, while the output buffer should be mutable. Having both a mutable and immutable reference to the same memory location violates the Rust borrowing rules and is therefore not possible.

As explained in the unsafe Rust section, Rust has raw pointers for which the borrowing rules do not apply. Therefore raw pointers can have multiple mutable and immutable pointers to the same location. In addition, it is possible to create a raw pointer to a fat pointer; this pointer only consists of the memory location and thus only takes up one register. Raw pointers are also not guaranteed to point to valid memory, can be null, and do not affect any other values when they go out of scope (i.e., no automatic drop). By loosening these restrictions, raw pointers allow for interoperation with other languages, hardware or, in some cases, allow for better performance of Rust code. Dereferencing (e.g., indexing the value) a raw pointer requires the use of `unsafe`. However, creating a raw pointer or passing a raw pointer to another function does not require `unsafe`, as only dereferencing a raw pointer can lead to undefined behavior. Using raw pointers in scenarios such as the one described above, the number of registers needed to pass two buffers of the same size, unknown at compile-time, can be reduced by one. Furthermore, this also allows for an immutable and mutable pointer to the same buffer, reducing the amount of memory needed to perform the cryptographic function.

Rust also provides what is known as composite types, which are build-up of primitive data types. These are structs, enums, and unions. The memory layout of these types is dependent on the primitive types they contain. The memory layout is referred to as their representation. By default, these composite types have the `default` Rust representation. This representation does not provide any guarantees about the memory layout. To interface with the C language, the programmer can use the C representation for these types:

```
#[repr(C)]
struct ElementLocation {
    address: u64,
    index: u8,
}
```

This guarantees that the size and alignment of the struct are the same as in C. We can also use the C representation to provide guarantees about the memory layout of composite types. This is useful when performing operations that make assumptions about the memory layout of the type.

Rust also has the `transparent` representation. This representation can only be applied to a struct with one field whose size is greater than 0 (and any number of zero-sized fields with an alignment of 1 byte). This ensures the memory layout of the struct is that of the field it contains. It also ensures this type follows the same ABI as the field it contains, i.e., the struct is passed in the same way to a function as the single field would have been passed.

```
#[repr(transparent)]
struct Float {
    num: f32,
}
```

For example, the struct above contains a single floating point number, on `x86_64` this would normally be passed using the normal registers as this is a struct. However, when using the transparent representation, it gets passed using the floating-point registers as is expected for a floating-point number.

3.2 Jasmin ABI

The Jasmin compiler outputs assembly files for the target machine architecture. An assembler then uses these assembly files to generate object code. The entry point of the Jasmin code is marked with the `export` keyword. Jasmin performs no name mangling of function names. Only the entry function marked with `export` remains visible in the final object code. Jasmin also does not export any of the global variables.

Like Rust, Jasmin uses the `System V ABI` on Linux `x86_64`. Thus it uses RDI, RSI, RDX, RCX, R8, R9 to pass arguments. As Jasmin supports SIMD instructions, on Intel x86 CPU's vector registers xmm0 up to xmm7 can be used to pass arguments. The Jasmin export function currently does not support stack arguments. Therefore when calling Jasmin code from external, only registers can be used as arguments, limiting the number of arguments that can be passed to Jasmin up to at most *six* 64-bit general-purpose registers and *eight* 128-bit vector registers.

In Jasmin, types passed to or returned from the export function are restricted to the unsigned integers `u8`, `u16`, `u32`, `u64`. When there is support for `AVX/AVX2`, the u128 and u256 types can also be used. Jasmin does not have a special type for pointers; on `x86_64` pointers to Jasmin are passed using u64. Jasmin treats the types u128 and u256 as vector types, which will therefore use the vector registers. All other types use general-purpose registers. Jasmin can return values in RAX, RDX, xmm0 and xmm1.

Jasmin only allocates memory on the stack. This memory is allocated on function entry and freed on function exit. When the programmer wants to make a change to a variable, made in Jasmin, visible to the caller of a Jasmin export function, this variable needs to be returned by this export function. Jasmin can also make changes to contents on the heap, but only through the pointers passed to it in an export function.

```
export fn add_three(reg u32 num, reg u64 bar)
{
    reg u32 foo;
    foo = num;
    foo += 3;

    [bar] += 3;
}
```

As the value `foo` is not returned from this function, the changes made by Jasmin are lost on function exit. As the pointer `bar` is provided by the caller, the changes made to this pointer will be visible to the caller.

Chapter 4

Rust and Jasmin Interoperation

To benefit from the high-level features and the ecosystem Rust provides while achieving maximum efficiency and verifiable secure cryptographic code, the goal is to interoperate between Rust and Jasmin.

The approach we take to call Jasmin from Rust is replacing the object file of a Rust Rlib with a Jasmin object file. First, the function to be converted to Jasmin is moved to a separate Rust file and compiled as an Rlib file. Then the Jasmin code is written for this function and compiled to an object file. By replacing the Rust object file with the Jasmin object file and keeping the metadata, the Jasmin code can be called without relying on FFI, and therefore there is no need for `unsafe` Rust. A high-level overview of this process can be seen in Figure 4.1. In this chapter, we will describe this approach in more detail. First, the matches and mismatches between the ABI of Rust and Jasmin are outlined. Then we describe a step-by-step approach to interoperate between Rust and Jasmin and introduce a tool to automate a part of these steps. Finally, we consider the safety of this approach.

4.1 Interoperation

By default, the Rust compiler performs symbol mangling; therefore, Rust will expect the name of the function it calls to be mangled. However, as Jasmin does not perform symbol mangling, calling Jasmin from Rust will result in errors during linking. Therefore the `#[no_mangle]` attribute has to be added to the Rlib function; this prevents Rust from mangling the function name.

As both Jasmin and Rust use the same calling convention, the first six arguments that fit in a general-purpose register are passed in registers: `RDI`, `RSI`, `RDX`, `RCX`, `R8` and `R9`. In Rust, when there are more than six arguments, they are passed via the stack. Jasmin currently does not support passing arguments

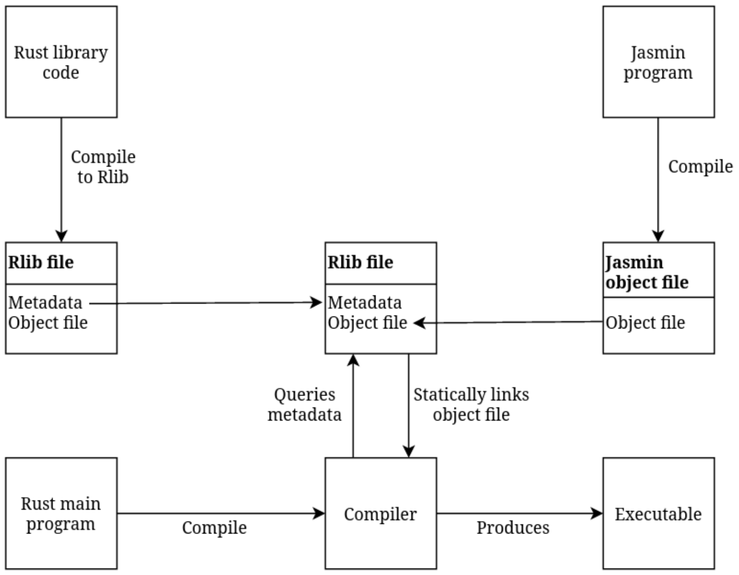


Figure 4.1: Rust to Jasmin overview

to the entry function via the stack; this limits the number of arguments that can be passed from Rust to Jasmin to six.

Rust has a rich type system, which in addition to the primitive types, supports user-defined composite types: structs, enums, and unions. Jasmin only has support for the types `u8`, `u16`, `u32`, `u64`, `u128`, `u256`. We only consider a subset of types as function arguments from Rust to Jasmin. Rust does not support `u256` value by default. Although Rust supports `u128` as a primitive type, Rust does not use a vector register for this type by default. Using u128 and u256 as arguments or return values is therefore not considered. The following unsigned integers from Rust are supported: u8, u16, u32, u64, usize. In addition to this, immutable and mutable references to arrays (`&[T; N]`/`&mut [T; N]`) and slices (`&[T]`/`&mut [T]`) are also supported. There is also support for raw pointers (`*const`/`*mut`) as this allows for creating a raw pointer to a fat pointer, which only needs one register instead of two (as explained in the previous section). For return types from Jasmin to Rust, the following Rust types are supported: `u8`, `u16`, `u32`, `u64` and `usize`. This subset of types is sufficient for writing typical cryptographic routines. The Rust types and their corresponding Jasmin types are shown in Table 4.1.

When passing a reference to a slice from Rust to Jasmin, this reference uses two registers. The Jasmin entry function should thus use two registers (as indicated in Table 4.1) when it receives a reference to a slice from Rust. When multiple buffers of the same length, unknown at compile-time, are passed to Jasmin, the programmer can use raw pointers to reduce the number of needed registers. This allows for more flexibility when dealing with the limit of at most

Rust type	Jasmin type
u8	reg u8
u16	reg u16
u32	reg u32
u64	reg u64
usize	reg u64
&/&mut [T; N]	reg u64
&/&mut [T]	reg u64, reg u64
const/*mut	reg u64

Table 4.1: Rust types and their corresponding Jasmin type

six register arguments.

Now we describe a step-by-step approach to interoperate between Rust and Jasmin. We assume the workflow of a programmer that has written their program in Rust and now wants to convert one function to Jasmin. A simple example of a Rust function that increases each element in a slice by its position is used to demonstrate the steps needed for Rust and Jasmin to interoperate. The programmer has written the following Rust code in the file `example.rs`:

```
fn main() {
    let mut input = [1,2,3,4];
    increase_by_pos(&mut input[..]);
    println!("result:␣{:?}", input);
}

fn increase_by_pos(input: &mut [u64]) {
    for i in 0..input.len() {
        input[i] += i;
    }
}
```

In the `main` function, an array is created with four elements. A mutable reference to a slice[1] is passed to the `increase_by_pos` function. After calling the function, the resulting array is printed. The programmer wants to convert the `increase_by_pos` function to a Jasmin function. First, we create a directory, named `jasmin`, for the Jasmin code to reside and a separate Rlib file, named `increase_by_pos.rs`.

```
    $ mkdir jasmin
    $ cd jasmin
    $ touch increase_by_pos.rs
```

[1]Here, the size of the array is known at compile-time, it would thus be possible to explicitly encode the size in the type of the receiving function. This is not done as a mutable reference to a slice serves as a more illustrative example.

The `increase_by_pos` function is moved to the Rlib file to generate the correct metadata. The `#[no_mangle]` attribute is added to prevent the Rust compiler from encoding a mangled name in the metadata. The function body is left empty since this is enough for Rust to encode the function in the metadata.

```
#[no_mangle]
pub fn increase_by_pos(input: &mut [u8]) {

}
```

Next, the programmer writes the Jasmin code in `increase_by_pos.jazz`:

```
export fn increase_by_pos(reg u64 input, reg u64 len) {
    reg u64 i;
    i = 0;
    while (i < len) {
        [input+8*i] += i;
        i += 1;
    }
}
```

A `u64` register is created as a counter for the while loop. While the counter is less than the length, the slice is indexed at each position and increased by the counter. The counter is multiplied by eight as each entry in the array takes up 8 bytes, and the array is indexed at byte level. Note that the Rust function only has one argument, but the Jasmin function has two. This is due to how the reference to a slice is passed in Rust (as explained previously). The programmer compiles the Jasmin function with the Jasmin compiler. This generates the assembly code, which is assembled to an object file using clang:

```
    $ <path_to_jasmin_compiler>/jasminc increase_by_pos.jazz
        -o increase_by_pos.s
    $ clang increase_by_pos.s -c -o increase_by_pos.o
```

When compiling the Rlib file, the compiler will output a warning about the unused variable `input`, but will still compile the program. After compilation the programmer extracts the metadata and packs it together with the Jasmin object file to create an Rlib file that can be called from Rust:

```
    $ rustc --crate-type=rlib increase_by_pos.rs
    $ ar -x libincrease_by_pos.rlib
    $ rm increase_by_pos.increase_by_pos.*
    $ rm libincrease_by_pos.rlib
    $ ar -crs libincrease_by_pos.rlib increase_by_pos.o
        lib.rmeta
```

The programmer now imports the Rlib file so that he can use the Jasmin function in `example.rs`. To import the Rlib the external crate `increase_by_pos` is defined and the function `increase_by_pos` is imported from this crate.

```rust
extern crate increase_by_pos;
use increase_by_pos::increase_by_pos;

fn main() {
    let mut input = [1,2,3,4];
    increase_by_pos(&mut input[..]);
    println!("result:␣{:?}", input);
}

//fn increase_by_pos(input: &mut [u64]) {
//    for i in 0..input.len() {
//        input[i] += i;
//    }
//}
```

The programmer now compiles and runs the program. To tell `rustc` where to find the external crate during the linking phase, the `-L` flag is used.

```
    $ rustc example.rs -L /jasmin
    $ ./example
```

Using the approach described above, we can call the Jasmin code from Rust.

Starting with just the Rust code, the following steps are observed in the process above:

1. Determine which functions need to be rewritten in Jasmin
2. Move these functions to a separate Rlib file
3. Create Jasmin functions
4. Write Jasmin functions
5. Compile Rlib files
6. Compile Jasmin files
7. Replace object file in Rlib with Jasmin object file

Automating part of this process simplifies calling from Rust to Jasmin. For this purpose, we create a Python tool named `jasminify`[2]. `jasminify` assumes the same workflow as above, i.e., the programmer wants to convert existing Rust code to Jasmin.

`jasminify` performs the process in two phases, with the corresponding commands `generate` and `build`. In the first phase, the tool scans the current directory for Rust files, and in each Rust file searches for functions annotated with the `"// Jasmin"` comment. It comments out all the annotated functions and moves them to Rlib files in a new `jasmin` directory. The Rlib files are

[2]https://gitlab.com/Jur/jasminify

automatically imported into the original Rust file so that the Rust program can call the functions from the original Rust file. Next, `jasminify` generates the corresponding Jasmin function stubs for the functions in the Rlib files. During the generation of the Jasmin stubs, `jasminify` performs several validation steps, which we will explain in the next section. The programmer now writes the Jasmin code for each of the generated Jasmin function stubs. In the second phase, the tool compiles the Jasmin functions and assembles them into object files. The Rlib file is compiled, and the object files in the Rlib are replaced with the Jasmin object files. Now the programmer compiles and runs the Rust program.

We now demonstrate `jasminify` using the same example as above.

```
    fn main() {
        let mut input = [1,2,3,4];
        increase_by_pos(&mut input[..]);
        println!("result:␣{:?}", input);
    }

    // Jasmin
    fn increase_by_pos(input: &mut [u64]) {
        for i in 0..input.len() {
            input[i] += i;
        }
    }
```

The function `increase_by_pos` is annotated with the `"// Jasmin"` comment to indicate to `jasminify` that, this functions should be converted to a Jasmin function. Now `jasminify` is executed with the generate command.

```
    $ python jasminify generate
```

`jasminify` automatically generates the Rlib file `example_increase_by_pos.rs` and the Jasmin file `example_increase_by_pos.jazz` in the `jasmin` directory. The generated Jasmin function stub is listed below:

```
//   slice: &mut[u64]
//   slice name: input
//   slice length: input_len
//   jasminc -checksafety -safetyparam input;input_len
export fn increase_by_pos(reg u64 input input_len)
{
    reg u64 i;
    i = 0;
    while (i < input_len) {
        [input+8*i] += i;
        i += 1;
    }
}
```

Note that `jasminify` added comments above the `increase_by_pos` function. This informs the programmer what Rust types are being passed to Jasmin. In this case, because the type being passed is a slice (`&mut [u64]`), it is expanded to two arguments, `input` and `input_len`. For slices `jasminify` also automatically specifies which safety parameters the programmer can pass to the safety checker to help with the Jasmin program's safety checking. We will explain this in the next section.

After completing the Jasmin function, we run the following command to compile the Jasmin code and Rlib:

```
    $ python jasminify build compiler=<path_jasmin_compiler>
```

The `compiler=` option is used to specify the path to the Jasmin compiler, which is used to compile the Jasmin files. Now we can compile the `example.rs` program and run it.

```
    $ rustc example.rs -L /jasmin
    $ ./example
```

4.2 Interoperation safety

Both Rust and Jasmin provide mechanisms to establish memory safety, and the safety guarantees they provide largely overlap. There are, however, some points the programmer should keep in mind when using the approach described above. These are discussed below.

The Rust compiler only queries the metadata corresponding to the Rust entry function; the Jasmin entry function can deviate from this. More specifically, the assembly generated code of the calling function in Rust should match the number of arguments and the types of the Jasmin entry function. As both Rust and Jasmin use the `System V AMD64 ABI`, the same registers will be used (up to a maximum of six arguments). When using a fat pointer (e.g., a pointer to a slice), Rust passes the argument using two registers; the Jasmin entry function should account for this.

`jasminify` ensures that these safety problems related to the ABI are prevented by performing several validation steps when generating the Jasmin stubs. As discussed earlier, only a subset of Rust types are allowed to be passed from Rust to Jasmin, `jasminify` verifies that only these allowed types are used. As the Rust type system is more expressive than Jasmin's, the programmer must translate Rust types to the correct Jasmin type (see Table 4.1). `jasminify` ensures that the correct Jasmin types are used. `jasminify` automatically expands Rust slices to two Jasmin register arguments. The Jasmin export function currently supports a maximum of six arguments due to Jasmin only accepting arguments via registers. `jasminify` ensures that a maximum of six arguments are used. Although Jasmin and Rust support returning more than one argument via registers, we only consider one return argument. `jasminify` prevents more than one return argument from being used. By performing these validation steps, `jasminify` ensures that the Jasmin code is called in the right way; this takes care of one part of the safety considerations the programmer should keep in mind.

Jasmin does not differentiate between the notion of immutability and mutability for externally provided arguments. Jasmin may therefore change the contents of an immutable variable, invalidating the safety invariant of Rust. When creating an immutable variable in Rust, it is impossible to assign a value to it again. Although Jasmin can change the contents of this variable when passed to Rust, the actual memory location in Rust where this variable is stored is never touched. Therefore these changes are not visible to Rust. Even when returning the value to Rust, assigning a returned value to an immutable variable is impossible. In this way, it is thus not possible for Jasmin to mutate immutable Rust variables. Jasmin can, however, changes the contents of memory declared immutable in Rust through pointers passed to the export function. When passing pointers to immutable memory to Jasmin, the programmer has to ensure that writing to memory only happens through mutable pointers. Furthermore, EasyCrypt provides a way to prove that values are not mutated; the programmer can use this approach to prove that the Jasmin code adheres to the immutability constraints of Rust.

Another problem is that Jasmin can change a pointer to an array and return this value to Rust. Jasmin has no understanding of the heap memory layout, so when returning a pointer from Jasmin, there are no guarantees that the pointer points to valid memory. This will be problematic if Rust uses this pointer. As already mentioned, the returning of pointers is not considered. `jasminify` ensures that only unsigned integers are returned from Jasmin to Rust.

To prevent data races, when dealing with multiple threads, Rust provides guarantees that prevent shared mutable data or requires synchronization to manage access to shared mutable data. Jasmin does not support multi-threading (i.e., Jasmin can not spawn additional threads) and therefore is not able to invalidate the safety guarantees provided by Rust.

Jasmin does not allocate or free on the heap; therefore, Jasmin does not interfere with the Rust heap. Since both Rust and Jasmin ensure memory safety independently, the operations performed by Jasmin and Rust are guaranteed not to result in memory errors; the one exception is if the Rust caller does not ensure the safety preconditions required by Jasmin. Jasmin can not know if the values passed to the entry function are valid. For this reason, the Jasmin safety checker reports which memory bounds should be valid; the programmer should verify these bounds.

We run the Jasmin safety checker on the `increase_by_pos` Jasmin example program above:

```
jasminc -safetyconfig range_config.json -checksafety
    example_increase_by_pos.jazz -safetyparam "input;input_len"
```

We pass a safety configuration file to the Jasmin safety checker using the `-safetyconfig` option. This is done to help the Jasmin safety checker with the safety analysis of our program. In this case, we specify a bound on the maximum value of `input_len`. The maximum size in bytes of a slice and array in Rust is $2^{47} - 1$. As each entry in the `&mut [u64]` slice points to 8 bytes, the maximum value of `input_len` is $(2^{47}/8) - 1 = 17592186044415$. We specify this upper bound as follows in the `range_config.json` file:

```
{
  "input_range": {
        "input_len": { "min": "0", "max": "17592186044415" }
    }
}
```

The Jasmin safety checker produces the following output:

```
*** No Safety Violation

Memory ranges:
  mem_input_len: [0; 0]

* Rel:
{inv_input_len  ≤  17592186044415, inv_input  ≤
18446744073709551615, inv_input  ≥  0, 8 · inv_input_len  ≥
mem_input, mem_input  ≥  0}
mem_input  ∈  [0; 140737488355320]

* Alignment: input 64;
```

The Jasmin safety checker reports no safety violations, meaning the code is memory safe, and the program terminates if the safety preconditions output by the safety checker are satisfied. The size of the allocated memory region pointed to by the `input` pointer depends on the value of `input_len`; therefore, Jasmin generates conjunctions of linear inequalities that need to hold for the allocated memory region.

In this case, it follows from the conjunctions of linear inequalities that $0 \leq mem\_input \leq 8 \cdot inv\_input\_len$, meaning that the `input` pointer must point to an allocated memory region of length $8 \cdot input\_len$. The upper bound on the `input` pointer is $17592186044415 \cdot 8 = 140737488355320$, which is expected as this is the maximum value at which the `input` pointer can be indexed. Since we pass a slice type (`&mut [u64]`) from Rust to Jasmin, both `input` and `input_len` are part of the same Rust type. The Rust type system guarantees that `input_len` exactly matches the number of elements pointed to by the `input` pointer and that this number is not larger than 17592186044415. The alignment specifies that the `input` pointer should be aligned to 64 bits. In Rust, references are equal to the word size; a pointer on `x86_64` is aligned to 64 bits. The safety preconditions of the Jasmin safety checker are thus satisfied.

When calling Jasmin from Rust, the programmer's responsibility is to check that the entry function and Rust function match, Jasmin does not mutate an immutable Rust value through a pointer, and the safety preconditions of the

Jasmin safety checker are satisfied. If these all hold, then the calling of Jasmin from Rust is guaranteed not to invalidate the safety guarantees of both languages.

Chapter 5

Case Study

The previous chapter gave a high-level overview of how the calling of Jasmin from Rust works and discussed the safety of this approach. This chapter demonstrates the interoperation process by replacing the assembly and C code of two algorithms (`x25519` and `ChaCha20`) in the Ring Rust library with Jasmin code.

5.1 Rustls

Rustls is a TLS library written in Rust [8]. Rustls is a modern library that aims to provide good cryptography security while not supporting obsolete or unsafe TLS features (e.g., RC4). For cryptography, Rustls relies on the Ring crypto library providing general-purpose cryptography implementations [27]. Most of the Ring crypto library code is based on BoringSSL[1], and it uses a mix of Rust, C, and assembly. Critically the low-level crypto routines are written in either C or assembly. This makes it a good candidate to replace some of the existing crypto code with Jasmin code.

For reproducibility purposes, the Rust version is set to the latest `rustc` version, with which we tested the interoperation:

```
    $  rustup toolchain install 1.51.0
    $  rustup default 1.51.0
```

The Ring library can be found on GitHub, from which it is cloned to edit the contents of the library:

```
    $ git clone https://github.com/briansmith/ring.git
```

To use the same version of the Ring library as in this case study, the git `checkout` command is used. This allows us to move back to a specific commit by using the commit hash. In this case, the hash of the latest commit when we wrote this case study.

```
    $ git checkout c263876eb52a5a5827d915efbdc45706197d2fdb
```

[1]https://boringssl.googlesource.com/boringssl

The Jasmin code of the algorithms replaced is introduced by Almeida et al [2] and can be found on Github[2]. Although we use existing Jasmin implementations to replace the existing code, we will write the case study using `Jasminify`. Thus, it is assumed that we still have to write the Jasmin code. This is done as this workflow is the most illustrative. The resulting code of this case study is published on GitLab[3].

5.1.1 x25519

`x25519` is an elliptic-curve Diffie Hellman (ECDH) function introduced by Bernstein in 2006 [6], that uses Curve25519 and provides 128-bit security. The algorithm can be used to construct a shared secret key over an untrusted channel safely. Curve25519 is defined in Montgomery form as follows: $y^2 = x^3 + 486662x^2 + x$ which operates on field $\mathbb{F}_{255} - 19$ and uses a fixed base point B where $X_B = 9$. The public, private and shared keys are all encoded into 32-byte arrays.

The procedure for establishing a shared key using `x25519` is as follows. Both parties sample a random 32-byte string uniformly at random. From this, the private key is generated by a process known as clamping. Clamping clears the lower three bits, clears the most significant bit, and sets the second most significant bit [24]. Using scalar multiplication, they compute their 32-byte public key by multiplying the generated private key with the base point. After exchanging public keys over the untrusted channel, they multiply their secret keys with the public key of the other party. This results in a shared secret value. This shared value can be used in a key-derivation function to generate a shared symmetric key.

The basic operation of the `x22519` algorithm is scalar multiplication, once to generate the public key and once to compute the shared key. For scalar s and point P the multiplication is written as sP. Using the Montgomery form, this operation can be performed efficiently using only the x-coordinate of the point. This works well with the algorithm as the public key is defined to be the x-coordinate. This has the added benefit that the public keys are only 32-bytes, in contrast to 64-bytes if both the x and y coordinate of the point would have been used. For scalar multiplication, the Montgomery ladder algorithm is used. The simple structure of this algorithm makes it easy for implementers to ensure the computation happens in constant time.

The clamping process has two purposes. First, the clearing of the lowest three bits and clearing of the most significant bit prevent small-subgroup attacks. Second, setting the second most significant bit prevents attempts to optimize the Montgomery ladder by skipping the first iteration if this bit equals zero. The skipping of the first iteration results in a timing leak that an adversary could exploit [9], [17].

[2]https://github.com/tfaoliveira/libjc
[3]https://gitlab.com/Jur/ring/-/tree/casestudy

5.1.2 Replacing x25519

Below we show the steps to replace the `x25519` crypto primitive from the Ring library (written in C) with the Jasmin implementation.

The `x25519` code is located at: `ring/src/ec/curve25519/x25519.rs`. Inspecting the `x25519.rs` file we observe the following code that calls into extern code:

```rust
#[cfg_attr(
    not(all(not(target_os = "ios"), target_arch = "arm")),
    allow(unused_variables)
)]
fn scalar_mult(
    out: &mut ops::EncodedPoint,
    scalar: &ops::MaskedScalar,
    point: &ops::EncodedPoint,
    cpu_features: cpu::Features,
) {
    #[cfg(all(not(target_os = "ios"), target_arch = "arm"))]
    {
        if cpu::arm::NEON.available(cpu_features) {
            return x25519_neon(out, scalar, point);
        }
    extern "C" {
        fn GFp_x25519_scalar_mult_generic_masked(
            out: &mut ops::EncodedPoint,
            scalar: &ops::MaskedScalar,
            point: &ops::EncodedPoint,
        );
    }
    unsafe {
        GFp_x25519_scalar_mult_generic_masked(out, scalar,
            point);
    }
}
```

The function `GFp_x25519_scalar_mult_generic_masked` is defined with the extern C ABI and called surrounded by the `unsafe` keyword. The code for this function is located at `ring/crypto/curve25519` and written in C. This C implementation of scalar multiplication is a good candidate to replace with a Jasmin implementation. As Jasmin currently only supports the `x86_64` architecture, we introduce a new function `x25519_jasmin` that is only called when compiled for the `x86_64` architecture. This is done using the `cfg` attribute, which allows for conditional compilation. In this case, the call to the Jasmin function is only present in the compiled program when compiled for `x86_64`. We annotate this new function with the `// Jasmin` comment so `jasminify` can convert this to a Jasmin function.

```rust
    #[cfg_attr(
        not(all(not(target_os = "ios"), target_arch = "arm")),
        allow(unused_variables)
    )]
    fn scalar_mult(
        out: &mut ops::EncodedPoint,
        scalar: &ops::MaskedScalar,
        point: &ops::EncodedPoint,
        cpu_features: cpu::Features,
    ) {
        #[cfg(all(not(target_os = "ios"), target_arch =
            "arm"))]
        {
            if cpu::arm::NEON.available(cpu_features) {
                return x25519_neon(out, scalar, point);
            }

        #[cfg(all(target_arch = "x86_64"))]
        {
            return curve25519_mulx(
                out,
                scalar.bytes(),
                point);
        }

        extern "C" {
            fn GFp_x25519_scalar_mult_generic_masked(
                out: &mut ops::EncodedPoint,
                scalar: &ops::MaskedScalar,
                point: &ops::EncodedPoint,
            );
        }
        unsafe {
            GFp_x25519_scalar_mult_generic_masked(out, scalar,
                point);
        }
    }

    #[cfg(all(target_arch = "x86_64"))]
    // Jasmin
    fn curve25519_mulx(out: &mut [u8; 32], scalar: &[u8; 32],
        point: &[u8; 32]) {

    }
```

The `scalar` variable is defined as follows: `pub struct MaskedScalar([u8; SCALAR_LEN])`. In order to pass a reference to the array contained in the struct the following function is added to the implementation of `MaskedScalar` in `ring/src/ec/curve25519/scalar.rs`:

```
    pub const fn bytes(&self) -> &[u8; SCALAR_LEN] {
        &self.0
    }
```

We now use `jasminify` to generate the Rlib and Jasmin function:

```
    $ python jasminify.py generate
```

`jasminify` generates the following Rust and Jasmin file in the Jasmin directory:

```
#[no_mangle]
pub fn curve25519_mulx(out: &mut [u8; 32], scalar: &[u8; 32],
    point: &[u8; 32]) {

}
```

```
// out: &mut[u8;32]
// scalar: &[u8;32]
// point: &[u8;32]
export fn curve25519_mulx(reg u64 out scalar point)
{

}
```

We observe that `jasminify` has preserved the Rust types in the comment above the Jasmin function. Now we write the Jasmin code for the generated Jasmin function stub. After completing this step, we again call `jasminify`:

```
    $ python jasminify.py build
        compiler=<path_jasmin_compiler> --opts=-lea
```

The `--opts=` option allows for the passing of options to the Jasmin compiler. The `-lea` option is used to tell the Jasmin compiler to use the `lea` assembly instruction as much as possible instead of the `add` and `mull` instructions. This can, on some CPUs, result in less execution time as the decoding of the `lea` instruction occurs in a different stage of the CPU pipeline. `jasminify` produces an Rlib file, in which the object file is replaced with the Jasmin object file.

The Ring library is compiled with the Rust Cargo package manager. In this case a Cargo build script is defined in `ring/build.rs`, as Cargo must compile non-Rust code. For Cargo to find and link our Rlib file, the `cargo:rustc-link-search` instruction is added to the build script (as suggested by `jasminify`):

```
fn ring_build_rs_main() {
    use std::env;
    ----- SNIP ------
    // Search for link targets in the Rust ....
    println!("cargo:rustc-link-search=<path_from_build.rs>
        /src/ec/curve25519/jasmin");
    check_all_files_tracked()
}
```

Now we can use Cargo to build the Ring library:

```
    $ cargo build
```

The ring library now calls Jasmin instead of C for the `x25519` scalar multiplication function as the build is successful.

The Ring library comes with built-in tests that verify the crypto functions are working as expected. One of these test functions compares the output of the `x25519` scalar multiplication function with multiple predefined inputs and their corresponding expected outputs. We can run this test to give confidence that the Jasmin implementation is called correctly and returns the correct output. We run the `x25519` test can be run using the following command:

```
    $ cargo test agreement_tests
```

The test runs without any errors, indicating that replacing the existing code with the Jasmin code was successful.

5.1.3 ChaCha20

`ChaCha20` is a stream cipher designed by Bernstein as a variation on the `Salsa20` cipher [7]. It improves on `Salsa20` by having a better diffusion per round and increased speed on some platforms [5]. The `ChaCha20` variant considered here is specified in RFC 7539 for use in TLS [18]. The `ChaCha20` algorithm has as input a 256-bit key, a 32-bit counter, a 96-bit nonce, and a plaintext of arbitrary length.

Constant	Constant	Constant	Constant
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Nonce	Nonce	Nonce

Table 5.1: State of the ChaCha algorithm. The 16 32-bit words are divided into 4 groups by colors. Each quarter takes one of these groups as input during the column round

.

The `ChaCha20` cipher consists of 20 rounds, generating a 64-byte keystream block. One round is made up of four quarter rounds, which each operate on four distinct words of the state. The state comprises 16 32-bit words, consisting of 4 constant words, 8 words derived from the key, 1 word from the counter, and 3 words from the nonce. The state can be seen as a four-by-four matrix and is

Constant	Constant	Constant	Constant
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Nonce	Nonce	Nonce

Table 5.2: State of the ChaCha algorithm. The 16 32-bit words are divided into 4 groups by colors. Each quarter takes one of these groups as input during the diagonal round

.

shown in Table 5.1. A quarter-round performs addition (modulo 2^{32}), bitwise XOR, and a left n-bit rotation on the input of four 32-bit words. The rounds alternate between operating on columns (shown in Table 5.1) and diagonals (shown in Table: 5.2). As each round operates on four distinct words of the state, it is possible on SIMD-supporting platforms to store the state in vector registers. `ChaCha20` can then be used with SIMD operations to increase the performance. After performing the 20 rounds, the initial state is added to the final state (modulo 2^{32}), resulting in a 64-byte keystream block. This block is XOR'ed with 64 bytes from the plaintext to get the encrypted output. When there is more plaintext to be encrypted, the 20 rounds are repeated with the same constant words, key, nonce, and the counter is increased by one.

5.1.4 Replacing ChaCha20

Below we show the steps to replace the `ChaCha20` crypto primitive from the Ring library (written in assembly) with the Jasmin implementation.

The `Chacha20` code is located at `ring/src/aead/chacha.rs`. Inspecting the `chacha.rs` file, the following code that calls into extern code is observed:

```
impl Key {
    ---- SNIP ----

    #[inline]
    fn encrypt_less_safe(&self, counter: Counter,
        in_out: &mut [u8], src: RangeFrom<usize>) {
        #[cfg(any(
            target_arch = "aarch64",
            target_arch = "arm",
            target_arch = "x86",
            target_arch = "x86_64"
        ))]
        #[inline(always)]
        pub(super) fn chacha20_ctr32(
            key: &Key,
            counter: Counter,
            in_out: &mut [u8],
            src: RangeFrom<usize>,
        ) {
            let in_out_len = in_out.len()
                .checked_sub(src.start).unwrap();

            // There’s no need to worry if ‘counter‘ is
            // incremented because it is owned here and we
            // drop immediately after the call.
            extern "C" {
                fn GFp_ChaCha20_ctr32(
                    out: *mut u8,
                    in_: *const u8,
                    in_len: crate::c::size_t,
                    key: &[u32; KEY_LEN / 4],
                    counter: &Counter,
                );
            }
            unsafe {
                GFp_ChaCha20_ctr32(
                    in_out.as_mut_ptr(),
                    in_out[src].as_ptr(),
                    in_out_len,
                    key.words_less_safe(),
                    &counter,
                )
            }
        }
    }
    ---- SNIP ----
}
```

This code shows that the function `GFp_ChaCha20_ctr32` is defined and called using the extern C ABI. We can find the code this function calls in the `ring/crypto/chacha/asm` directory. Here multiple `ChaCha20` implementations are found for different CPU architectures, all written in assembly. This code makes for a good candidate to replace with Jasmin code.

Depending on the CPU features available, different assembly code is run to get the best performance. The Ring `ChaCha20` assembly implementation checks for availability of the `AVX` and `AVX2` features. Three separate Jasmin implementations are used, reference, `AVX`, and `AX2`, and called depending on the CPU features available. To call the right Jasmin function, we will check the supported CPU features in Rust. For this, the Ring crate provides the CPU (`ring/src/cpu.rs`) Rust module, which checks for supported CPU options by executing the `CPUID` instruction. Ring already provides the ability to check for the `AVX` feature. Support for checking for the `AVX2` feature is added.

```
#[cfg_attr(
    not(any(target_arch = "x86", target_arch = "x86_64")),
    allow(dead_code)
)]
pub(crate) mod intel {

    ---- SNIP ----

    pub(crate) struct Feature {
        word: usize,
        mask: u32,
    }
    #[cfg(target_arch = "x86_64")]
    pub(crate) const AVX: Feature = Feature {
        word: 1,
        mask: 1 << 28,
    };

    #[cfg(target_arch = "x86_64")]
    pub(crate) const AVX2: Feature = Feature {
        word: 2,
        mask: 1 << 5,
    };

    ---- SNIP ----

}
```

As Jasmin currently only supports `x86_64`, we can replace the assembly code of this architecture with Jasmin code. A new function `chacha20_jasmin` is written that is only introduced when compiled for the `x86_64` architecture. In this function, the available CPU features are checked, and the right Jasmin implementation is called.

```
impl Key {
    ---- SNIP ----

    #[cfg(any(
        target_arch = "aarch64",
        target_arch = "arm",
        target_arch = "x86",
        target_arch = "x86_64"
    ))]
    #[inline(always)]
    pub(super) fn chacha20_ctr32(
        key: &Key,
        counter: Counter,
```

```rust
        in_out: &mut [u8],
        src: RangeFrom<usize>,
    ) {
        let in_out_len = in_out.len()
            .checked_sub(src.start).unwrap();
        #[cfg(all(target_arch = "x86_64"))]
        {
            return chacha20_jasmin(
                in_out.as_mut_ptr() as *mut u64,
                in_out[src].as_ptr() as *const u64,
                in_out_len as u32,
                key,
                counter.0[1..4].as_ptr() as *const u64,
                counter.0[0]);
        }

        // There's no need to worry if `counter` is
        // incremented because it is owned here and we drop
        // immediately after the call.
        extern "C" {
            fn GFp_ChaCha20_ctr32(
                out: *mut u8,
                in_: *const u8,
                in_len: crate::c::size_t,
                key: &[u32; KEY_LEN / 4],
                counter: &Counter,
            );
        }
        unsafe {
            GFp_ChaCha20_ctr32(
                in_out.as_mut_ptr(),
                in_out[src].as_ptr(),
                in_out_len,
                key.words_less_safe(),
                &counter,
            )
        }
    }

    ---- SNIP ----
}
#[cfg(all(target_arch = "x86_64"))]
#[inline]
fn chacha20_jasmin(output: *mut u64, plain: *const u64,
    len: u32, key: &Key, nonce: *const u64, counter: u32)
    {
        if cpu::intel::AVX2.available(key.cpu_features) {
            return chacha20_avx2(output, plain, len,
                key.words_less_safe(), nonce, counter)
        }
        if cpu::intel::AVX.available(key.cpu_features){
            return chacha20_avx(output, plain, len,
                key.words_less_safe(), nonce, counter)
        }
        chacha20_ref(output, plain, len,
            key.words_less_safe(), nonce, counter)
    }
```

As these new functions are going to be converted to a Jasmin function, they are annotated with the `// Jasmin` comment[4].

```
// Jasmin
fn chacha20_ref(output: *mut u64, plain: *const u64, len: u32,
    key: &[u32; 8], nonce: *const u64, counter: u32) {
}

// Jasmin
fn chacha20_avx(output: *mut u64, plain: *const u64, len: u32,
    key: &[u32; 8], nonce: *const u64, counter: u32) {
}

// Jasmin
fn chacha20_avx2(output: *mut u64, plain: *const u64, len: u32,
    key: &[u32; 8], nonce: *const u64, counter: u32) {
}
```

Now we call `jasminify` to generate the Rlib and Jasmin function stubs:

```
    $ python jasminify.py generate
```

This generates the Rlib and Jasmin files for the reference, `AVX`, and `AVX2` functions in the Jasmin directory. We now describe the second step of replacing the algorithm for the `ChaCha20` reference implementation. The process is similar for `AVX` and `AVX2` implementations. Looking at the generated Rlib and Jasmin file for the reference implementation, the following code is observed:

```
#[no_mangle]
pub fn chacha20_ref(output: *mut u64, plain: *const u64,
    len: u32, key: *const u64, nonce: *const u64, counter: u32)
{

}
```

[4]As existing Jasmin implementations are used the way they pass function arguments is changed to match the Jasmin functions.

```
// output: *mutu64
// plain: *constu64
// len: u32
// key: &[u32;8]
// nonce: *constu64
// counter: u32
export fn chacha20_ref(reg u64 output, reg u64 plain,
    reg u32 len, reg u64 key, reg u64 nonce, reg u32 counter)
{

}
```

Now we write the code for the Jasmin reference implementation. After this is done, we run `jasminfiy` to build the final Rlib:

```
    $ python jasminify.py build
        compiler=<path_jasmin_compiler>
```

For Cargo to find and link our Rlib file, the `cargo:rustc-link-search` instruction is added to the build script.

```
fn ring_build_rs_main() {
    use std::env;
    ----- SNIP ------
    // Search for link targets in the Rust
    println!("cargo:rustc-link-search=<path_from_build.rs>/
        ring/src/aead/jasmin");
    check_all_files_tracked()
}
```

Now we use Cargo to build the Ring library:

```
    $ cargo build
```

As the build is successful, the Ring library now calls Jasmin instead of assembly for the `ChaCha20` encryption algorithm. The Ring library comes with built-in tests that verify the crypto functions are working as expected. One of these test functions verifies the output of the `ChaCha20` function with predefined input and output pairs. We can run this test to give confidence that the Jasmin implementation is called correctly and returns the correct output. We run the `ChaCha20` test using the following command:

```
    $ cargo test chacha20_test_default
```

The test runs without any errors, indicating that the replacing of the code was successful.

We repeated the process above for the `AVX` and `AVX2` Jasmin implementations[5]. This, however, resulted in an error when running the Ring test for the

[5] As there were constants with the same name in both AVX and AVX2 the names of the constants in AVX2 were changed to prevent linking errors.

ChaCha20 function. This is because Ring allows for partially overlapping input and output buffers, where output pointer <= input pointer. This is verified during the ChaCha20 test case by testing for offset values between 0 and 259. Working with buffers that completely overlap (offset = 0) or are completely disjoint is not a problem, but working with buffers that partially overlap results in an error. This happens as the AVX and AVX2 implementation store the ciphertext in the output buffer in two rounds in an interleaved fashion.

For example, consider the AVX implementation with a length of 256 and an offset of 1 (`input pointer = output pointer + 1`). After performing the ChaCha20 rounds, the plaintext gets XOR'ed with the keystream and stored in the output buffer in two separate rounds. In the first round, the encrypted text is written to the output pointer at ranges 0-31, 64-95, 128-159, 192-233. In the following round, data is read from the input pointer at intervals 32-63, 96-127, 160-191, 234-255. But as `input pointer = output pointer + 1`, from the perspective of the output pointer, the following ranges are read 33-64, 96-128, 161-192, 234-255. It thus reads bytes 64, 128, and 192, which are assumed to be plaintext but have already been overwritten with chipertext in the first round. These bytes are XOR'ed with the keystream again and then written to 32-63, 96-127, 160-191, 234-255. This results in wrong output for bytes 63, 127, and 191. It is important to note that this is not a safety problem but a correctness problem (i.e., the input and output behavior of AVX and AVX2 implementations does not match that of the reference implementation). The Ring library is more permissive by allowing the input and output pointer to partially overlap, and therefore does not satisfy the contract for which the AVX and AVX2 implementations were proven correct.

To fix this issue, we changed the AVX and AVX2 implementations to store the plaintext read in the second round on the stack before the encrypted output of the first round is written to the output pointer. In the second round, the plaintext is then read from the stack and XOR'ed with the keystream. By applying this change, the ChaCha20 test runs without any errors for the AVX and AVX2 implementations[6].

5.2 Benchmarks

We perform a performance evaluation of both the ChaCha20 and x25519 Jasmin implementations. For this evaluation, we use the Rust `criterion` benchmarking crate version 0.3.5 [11], a statistics-based micro-benchmarking tool used to detect performance regressions and improvements between the current and the previous version of the code.

The benchmarks were run on an Intel i7-8650U processor clocked at 1.90GHz with Turbo Boost and Hyper-threading disabled, running kernel release 5.13.12-

[6]We did not redo the correctness proof for the changes made to the Jasmin AVX and AVX2 implementations. Redoing this proof should be straightforward, and we leave this for future work.

arch1-1. We compiled the Ring library with rustc version 1.51. We compiled the non Rust parts with `gcc` 11.1[7].

The Jasmin implementations are compared against the existing Ring implementations. For `x25519` the `x25519_ecdh` function is benchmarked. The results for `x22519` are shown in Table 5.3. For `x25519`, we note a significant perfor-

Ring x25519	Jasmin x25519
82.564 us	51.937 us

Table 5.3: Average time in microseconds of both algorithms based on over 120k iterations of both implementations

mance increase (37%). This is expected as the Ring library implementation of `x22519` is implemented in C instead of assembly. Furthermore, the Jasmin implementation also directly uses the `MULX`, `ADOX` and `ADCX` instruction introduced by Intel for big-integer arithmetic [21].

To evaluate the performance of `ChaCha20` the execution time of the `encrypt_less_safe` function is measured. We only compare the Jasmin `AVX2` implementation against the Ring implementation. This is done as the system the evaluation is performed on supports `AVX2`, and the existing Ring implementation will default to using `AVX2`. The results for `ChaCha20` are shown in Table 5.4 for different input lengths. Figure 5.1 shows the throughput.

#bytes	Ring ChaCha20	Jasmin ChaCha20
256	319.82 ns	237.05 ns
512	324.01 ns	319.03 ns
1024	643.84 ns	642.45 ns
2048	1254.3 ns	1252.9 ns
4096	2507.7 ns	2506 ns
8192	5027.6 ns	5019.8 ns

Table 5.4: Average time in nanoseconds of both algorithms for varying input sizes, based on over 1.8 million iterations for all runs

The performance measurements of both `ChaCha20` implementations do not show any significant differences. This is to be expected as the Ring implementation already relies on optimized assembly that uses `AVX2` instructions. Only for the small size of 256 do we notice the Jasmin implementation being faster (about 26%). We suspect this difference is present because the Jasmin implementation uses a different strategy for inputs of less than 257 bytes.

The `x25519` implementation improves performance, while the `ChaCha20` has similar performance. We note that both the Jasmin implementations have cor-

[7]The benchmark code can be found at https://gitlab.com/Jur/ring/-/tree/casestudy/benches

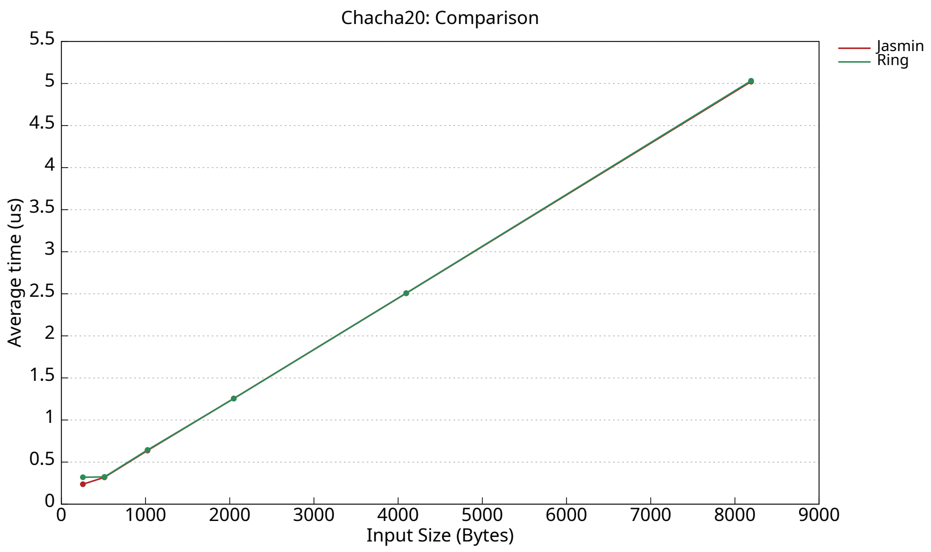


Figure 5.1: Throughput

rectness proofs. Although this can also be achieved for the existing Ring implementations, Jasmin provides this all in one framework and makes it easier to iterate new improvement ideas.

5.3 Safety

Now both algorithms have been replaced with Jasmin code and were tested using the tests provided by Ring; we will consider the safety of both implementations.

5.3.1 X25519

To verify the Jasmin x25519 implementation we run the Jasmin safety checker:

```
    $ ../jasminc.native -checksafety
    x25519_curve25519_mulx.jazz -safetyparam
    "curve25519_mulx>;"
```

Using the `safetyparam` option, we specify which function to analyze. The output of the Jasmin safety checker is listed below:

```
Analyzing function curve25519_mulx

*** No Safety Violation

Memory ranges:
  mem_out: [0; 32]
  mem_scalar: [0; 32]
  mem_point: [0; 32]

* Rel:
 ⊤

* Alignment: out 64; scalar 64; point 64;
```

We observe that the memory access to the pointers is done between 0 and 32 bytes, which is as expected. Both `out` and `point` are defined as an `EncodedPoint` looking for the definition in the Rust code for the `EncodedPoint` type the following is found in the file `ops.rs`:

```
    pub type EncodedPoint = [u8; ELEM_LEN];
    pub const ELEM_LEN: usize = 32
```

A reference to the EncodedPoint is passed to Jasmin. We are thus sure that both `point` and `out` point to 32 bytes of memory. `scalar` is defined in the struct `Scalar`:

```
#[repr(transparent)]
pub struct Scalar([u8; SCALAR_LEN]);
pub const SCALAR_LEN: usize = 32;
```

When passing this type to Jasmin, a reference to the inner value (`[u8; SCALAR_LEN]`) is passed to Jasmin. We are thus guaranteed that this value points to 32 bytes of data.

The alignment requires that `scalar`, `out`, and `point` are aligned to 64 bits. Since references in Rust have the same alignment as usize on `x86_64`, this is the case.

Looking at the Jasmin code, we observe only one place where is being written to the mutable `out` pointer. No writes to the immutable `scalar` and `point` pointers are observed. Since Jasmin can only modify the Rust memory through these pointers, we are sure Jasmin does not invalidate the immutability constraints of Rust.

5.3.2 ChaCha20

To check the safety of the Jasmin `ChaCha20` implementation, we run the Jasmin safety checker on the Jasmin implementations. We only demonstrate this process for the `ChaCha20` reference implementation. The same process and reasoning hold for the `AVX` and `AVX2` implementations.

```
    $ ../jasminc.native -checksafety -safetyparam
        "plain;len:output;len" chacha_chacha20_ref.jazz
```

To help the Jasmin safety checker, the parameters `plain;len` and `output;len` are specified to get the bounds on memory access for the plaintext and output pointer. The output of the Jasmin safety checker is listed below:

```
*** No Safety Violation

Memory ranges:
  mem_len: [0; 0]
  mem_key: [0; 32]
  mem_nonce: [0; 12]
  mem_counter: [0; 0]

* Rel:
{mem_plain  ≥  0, inv_len  ≤  4294967295, inv_len  ≥
    mem_plain}
mem_plain  ∈  [0; 4294967295]

* Alignment: output 64; plain 64; key 32; nonce 32;
* Rel:
{mem_output  ≥  0, inv_len  ≤  4294967295, inv_len  ≥
    mem_output}
mem_output  ∈  [0; 4294967295]

* Alignment: output 64; plain 64; key 32; nonce 32;
```

Running the Jasmin safety checker on the code presents no safety violations. The value of `len` is not indexed, therefore the memory range is `[0; 0]`. The same holds for the `counter` value. The pointer to `Key` is indexed from 0 up to 31. Therefore the memory range is [0; 32] (inclusive on the left and exclusive on the right), meaning Jasmin expects the Key pointer to point to 32 bytes. For `nonce`, Jasmin expects it to point to 12 bytes specified by the memory range [0; 12]. Looking at the Rust code for the `Key` value, the following is found in `chacha.rs`:

```
pub struct Key {
    words: [u32; KEY_LEN / 4],
    cpu_features: cpu::Features,
}

pub const KEY_LEN: usize = 32;
```

When passing the `Key` to Jasmin, only the `words` member of the `Key` struct is passed. Thus, Jasmin receives a reference to `[u32; 8]`, which points to 32 bytes. The `nonce` is grouped with the counter in the `Iv` struct:

```
pub struct Iv([u32; 4]);
```

The first 4 bytes represent the `counter` and the following 12 bytes represent the `nonce`. The `nonce` value is passed to Jasmin in the following way:

```
counter.0[1..4].as_ptr() as *const u64,
```

Passing the `nonce` value in this way results in a pointer being passed to the last 3 `u32` values of the `Iv` struct. This equals the expected 12 bytes.

The `Rel` entry states the memory range through conjunctions of linear inequalities. From the entry it is observed that both the `plaintext` and `output` pointer are accessed anywhere between 0 and the value of `len`. The value of `len` is defined as follows in the Rust code:

```
let in_out_len = in_out.len().checked_sub(src.start).unwrap();
```

The pointer `in_out` refers to the input and the output buffer (only one buffer is passed to the `encrypt_less_safe` function). In this way, the `len` value is the same for both the input buffer and the output buffer. From this, we conclude that all accesses to the plaintext and output pointer happen on valid and initialized memory.

Looking at the Jasmin code, we observe that the values `key`, `len`, `nonce`, and `counter` are only read. The interesting cases are the `plain` and `output` references declared immutable and mutable, respectively. These references can point to the same or an overlapping memory range. This is not in line with the Rust safety rules, which require that at any given time, either *one* unique mutable reference can mutate a memory location or a memory location can be shared many times immutably. In contrast to Rust, Jasmin cannot affect the memory location of the objects allocated on the heap (e.g., by pushing something onto a buffer). Jasmin can only mutate the values to which the `plain` and `output` pointers point. Therefore having one immutable and one mutable reference, provided by the caller of the export function, to the same or overlapping memory location does not result in memory safety issues in Jasmin. Furthermore, since we have a mutable pointer to `output` we are sure that this pointer is not used anywhere else by the caller code. Rust can thus not modify the memory location of where the `output` and `plain` pointers point to.

In this way, it is guaranteed the code of both `x25519` and `ChaCha20` performs in-bounds array access, accesses only valid memory, is absent of division by zero, and that all variables are initialized. Both Jasmin implementations have also been proven absent of secret-leaking side channels and functionally correct [2]. We can now use the Ring library with Jasmin code for the `x86_64` platform.

Chapter 6

Discussion

6.1 Future work

We see several ways to improve the safety of the interoperation approach introduced in this thesis. First, the safety analysis is not fully automated yet. It is possible to extent `jasminify` to be aware of the Rust types being passed and check if the safety preconditions hold. Alternatively, the Jasmin safety checker could also be extended to accept and understand the Rust types of the calling function. This way, the Jasmin safety checker could validate if the safety preconditions are satisfied. Extending the Rust compiler to be aware of the Jasmin ABI, e.g., using `extern Jasmin` for Jasmin functions, would further simplify the interoperation between Rust and Jasmin. Tighter integration with the Rust compiler could also provide Jasmin with more information about the types it receives. This further simplifies the automatic reasoning of the Jasmin safety checker.

Furthermore, in this thesis, we only reason why the introduced approach preserves the safety guarantees of both languages. However, no formal proof that our reasoning holds is given. Giving a formal proof would show that programs using our approach do, in fact, preserve the safety guarantees of both Rust and Jasmin. This requires the formalization of Rust, which is being worked on by the RustBelt [14] and Oxide [30] projects.

These extensions would also justify the absence of the `unsafe` keyword in our approach, as this would fully eliminate the programmer's responsibility to ensure the code is safe.

We also see ways to improve the applicability of the interoperation approach. `jasminify` currently supports a limited set of types. `jasminfiy` could be extended to allow for more Rust types. It would be especially interesting to introduce support for Rust structs. The `#[repr C]` attribute could guarantee the struct layout in memory. Currently, the amount of arguments passed from Rust to Jasmin is limited to at most six. This is due to Jasmin not being able to accept arguments via the stack. For the algorithms studied in the case study, this did not turn out to be a problem. However, this might not hold for all algorithms. Therefore Jasmin could benefit from accepting arguments via the stack. This is not a straightforward step as it will complicate the memory model of Jasmin. Therefore, making it harder to reason about the safety of Jasmin programs. Furthermore, in this thesis, only the `x86_64` architecture is considered. Currently, Jasmin only supports the `x86_64` architecture, but support for the ARM and RISC-V architectures is being worked on. A similar approach to analyze the ABI and the safety as in this thesis can be used for these architectures.

6.2 Conclusion

This thesis introduced an approach to interoperate between Rust and Jasmin for the `x86_64` architecture. This approach makes it possible to call Jasmin functions from Rust in a safe manner without relying on the Rust `unsafe` keyword. In addition, a Python tool, `jasminify`, is introduced to make the process easy for the programmer and prevent errors. We explain how to use this approach safely, and in the case study, we demonstrate our approach to replace two algorithms of an existing project.

We hope to contribute to the safety of the software ecosystem by enabling developers to use the Rust programming language in combination with the Jasmin framework for cryptographic code. This work also serves as a stepping stone for creating formal proof about the interoperation approach. Furthermore, the same process followed in this thesis for the `x86_64` architecture can be used for other architectures.

References

[1] José Bacelar Almeida et al. "Jasmin: High-assurance and high-speed cryptography". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1807–1823.

[2] José Bacelar Almeida et al. "The last mile: High-assurance and high-speed cryptographic implementations". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 965–982.

[3] José Bacelar Almeida et al. "Verifying constant-time implementations". In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 53–70.

[4] Gilles Barthe et al. "Easycrypt: A tutorial". In: *Foundations of security analysis and design vii*. Springer, 2013, pp. 146–166.

[5] Daniel J Bernstein et al. "ChaCha, a variant of Salsa20". In: *Workshop record of SASC*. Vol. 8. 2008, pp. 3–5.

[6] Daniel J Bernstein. "Curve25519: new Diffie-Hellman speed records". In: *International Workshop on Public Key Cryptography*. Springer. 2006, pp. 207–228.

[7] Daniel J Bernstein. "The Salsa20 family of stream ciphers". In: *New stream cipher designs*. Springer, 2008, pp. 84–97.

[8] J. Birr-Pixton. *a modern TLS library in Rust*. https://github.com/rustls/rustls. Accessed on 20.08.2021.

[9] Wouter de Groot. *A Performance Study of X25519 on Cortex-M3 and M4*. 2015.

[10] *Guide to Rustc Development*. URL: https://rustc-dev-guide.rust-lang.org/.

[11] B. Heisler. *Criterion.rs*. https://github.com/bheisler/criterion.rs. Accessed on 15.08.2021.

[12] Bertrand Jeannet and Antoine Miné. "Apron: A library of numerical abstract domains for static analysis". In: *International Conference on Computer Aided Verification*. Springer. 2009, pp. 661–667.

[13] Ralf Jung. "Understanding and evolving the Rust programming language". PhD thesis. 2020.

[14] Ralf Jung et al. “RustBelt: Securing the foundations of the Rust programming language”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–34.

[15] Ralf Jung et al. *Safe systems programming in Rust: The promise and the challenge*. In: CACM (2020). To appear.

[16] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[17] MARTIN Kleppmann. *Implementing Curve25519/X25519: A tutorial on elliptic curve cryptography*. Tech. rep. Tech. rep., University of Cambridge, Department of Computer Science and ..., 2020.

[18] Adam Langley et al. “ChaCha20-Poly1305 cipher suites for transport layer security (TLS)”. In: *RFC 7905* 10 (2016).

[19] Nicholas D Matsakis and Felix S Klock. “The rust language”. In: *ACM SIGAda Ada Letters* 34.3 (2014), pp. 103–104.

[20] Microsoft. *A proactive approach to more secure code*. Accessed on 23.08.2021. URL: https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code.

[21] E Ozturk et al. “New instructions supporting large integer arithmetic on intel architecture processors”. In: *Intel white paper, reference 327831-001* (2012).

[22] Chromium project. *Memory safety*. Accessed on 23.08.2021. URL: https://www.chromium.org/Home/chromium-security/memory-safety.

[23] Qualys. *Qualys Security Advisory - The Stack Clash*. June 2017. URL: https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt.

[24] Peter Schwabe et al. “A Coq proof of the correctness of X25519 in TweetNaCl.” In: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 428.

[25] Matthew S Simpson and Rajeev K Barua. “MemSafe: ensuring the spatial and temporal memory safety of C at runtime”. In: *Software: Practice and Experience* 43.1 (2013), pp. 93–128.

[26] Gagandeep Singh, Markus Püschel, and Martin Vechev. “Fast polyhedra abstract domain”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017, pp. 46–59.

[27] B. Smith. *general-purpose cryptography in Rust*. https://github.com/briansmith/ring. Accessed on 27.08.2021.

[28] *The Rust reference*. Accessed on 23.08.2021. URL: https://doc.rust-lang.org/1.51.0/reference/.

[29] *The Rustonomicon*. Accessed on 07.11.2020. URL: https://doc.rust-lang.org/nomicon/races.html.

[30] Aaron Weiss et al. “Oxide: The essence of rust”. In: *arXiv preprint arXiv:1903.00982* (2019).