Serval

Integrating an existing symbolic verification framework with Rust

Master Semester Project

Expressing properties of libraries with finite interfaces within the Rust language

Filippo Costa

Supervisors :
Prof. Dr. Edouard Bugnion
Charly Castes

Data Center Systems Lab
EPFL, Lausanne, Autumn 2022

Acknowledgements

Often, saying thanks to people in this manner seems like a clichè. Nevertheless, my gratefulness goes to Prof. Ed Bugnion, which spent hours of his time sharing knowledge and food for thought, and to Charly Castes, that helped and supported me through the entire semester in any circumstance, from solving silly problems to discussing engaging ideas.

Contents

1 Introduction **1**
- 1.1 Serval . . . 1
- 1.2 Symbolic Execution . . . 1
- 1.3 Motivating example . . . 2
- 1.4 Scope of the project . . . 4

2 How to use Serval? **5**
- 2.1 Components . . . 5
- 2.2 Assumptions and Limitations . . . 6
- 2.3 Overview of the verification workflow . . . 6
- 2.4 Serval usage . . . 7
- 2.5 Issues encountered . . . 10

3 Integration in Rust **11**
- 3.1 Overview and high-level description . . . 11
- 3.2 Syntax . . . 11
- 3.3 Implementation . . . 16
- 3.4 Evaluation and limitations . . . 17

4 Conclusions **18**
- 4.1 Further ideas . . . 18

References **20**

A Appendix **21**
- A.1 Properties expressed in Racket . . . 21
- A.2 Properties expressed in Rust . . . 23
- A.3 Motivating Example . . . 24

1 Introduction

This first section will introduce Serval and the scope of the project. The second section will provide an overview of the initial verification environment, along with an explanation of how to adapt it to verify Rust code. The third section will describe the proposed language and methods to write lemmas and program properties directly in the Rust code.

1.1 Serval

Serval [1] is a verification framework, written in Racket, that provides a working environment to evaluate the behavior of programs. In particular, its declared scope is to be used with security monitors and system code in general. It leverages symbolic execution to verify or disprove program properties.

It comes with a set of functionalities and ideas. It has built-in support for RISC-V, eBPF (x86-32), and LLVM instruction sets and allows the development of their interpreters. It brings, besides the interpretation of machine-level programs, also some specific optimizations to the symbolic execution of that kind of paradigm.

While Serval does not explicitly target Rust, it could be used with it, because Serval verifies machine code and not high-level source code.

There exist also other verification tools specifically made for Rust, for example, Prusti [2]. The key difference between it and Serval is that the former directly operates on Rust code, thus it can exploit high-level assumptions and peculiarities of the language. On the other side, it lacks customization and flexibility regarding low-level constructs.

1.2 Symbolic Execution

Symbolic execution extends the standard execution model, referred also as "concrete" afterward, adding the concept of symbolic values. Concrete variables, during the execution, assume a specific value, that maybe can be updated and altered, and that can also interfere with the execution flow. This is the well-known concept of program execution on a real computer. On the other side, during symbolic evaluation, variables are not a specific value, but rather all the possible ones that are achievable during the execution. Step by step, values are added, along with a condition on input and other values, that represents when a symbolic value could assume a concrete value, based on the different paths that the execution flow may follow. In the end, a symbolic variable is a collection of conditions on input values and related values. The original definition of symbolic execution can be found in [3]. It allows the evaluation at once of all possible behaviors of a program. On the other hand, it comes with

increased memory and computation costs. If not made explicit by the context, we will refer to symbolic – and not to concrete – execution, interpretation, or evaluation. All of these terms have a similar meaning in our frame of reference.

Symbolic execution, in our context, can be interpreted as a generalization of unit testing. In fact, it helps to write more concise tests with a generally higher coverage compared to standard concrete unit testing.

When we use symbolic execution with `assertions`, the result is either a counterexample, that is a set of input values for which the `assertion` is proved wrong, or a successful result.

1.3 Motivating example

Let's assume that we have a global variable, `containers`, and a function, `init_container`. `containers` is an array of 32 structs, where each has an integer `id` field and other data. `init_container` receives as input an integer `container_id` and initializes a container in the array if one of them is empty (has 0 as `id`) and no other container with the same `id` exists. It returns `0`, if the container is created, or `1` otherwise. We want to prove two arbitrarily chosen properties, namely: (1) if the result is `0`, then there exists at least one element in the `containers` array such that `id` equals `container_id`, and (2), if a container with the same `id` already exists, then the result of `init_container` is `1` (it fails).

Along with writing a piece of code that we think satisfies the properties (see A.3.1), we have to define in Racket the two lemmas that represent the desired properties. We also need to integrate them into the testing framework such that they will be verified. With simple properties like (1) and (2), we accomplish this by writing some dozens of lines of Racket code. With more lemmas, the size of the Racket files obviously increases, and with that their intricacy.

An hypothetical definition for property (1) may be:

```
(define (lemma-1-prove-creation)
  (define-symbolic new_container_id i64)

  (if (bveq (@init_container new_container_id) (bv 0 i8))
    (let ()
      (define contains #f)
      (for ([c 32])
        (define current_id (mblock-iload (llvm:symbol->block 'containers) (list
        ↪  c 'id)))
        (set! contains (or (bveq new_container_id current_id) contains))
      )
      (assert contains)
    )
```

```
    (assert #t)
  )
)
```

The complete specification of both (1) and (2) is available in the appendix (A.3.2).

Then, we have to compile the program code to two different file formats, for code and memory structure, and at last, we have to run the Racket tests. The details of this procedure are explained in 2.4.2.

This comes, ideally, with two problems. The first is that it does not work out of the box with Rust. During the last few years, Rust affirmed itself to be a solid, memory-safe, and performant alternative to languages like C and C++ in low-level contexts. Having different verification frameworks available is then essential. The second is that writing code in Racket to express properties may be tedious and unsustainable in the long run, in particular when it comes to accessing memory and working with complex environments. It becomes evident in the specifications provided by Serval authors as examples [4]. Given the number of involved components, it is difficult to understand the structure of the lemmas before spending some time jumping between the program and them. It creates an undesirable obstacle that may hurt the adoption of this framework, or of formal verification in general, as a method to assess the quality of a program. Also, reviewing the verification files by a third party, not directly involved in the development, may become burdensome and not straightforward.

This project approaches these issues proposing a language that is similar to Rust and is mapped directly to Racket statements. The above (1) property would be expressed, directly in a Rust file, as:

```
"lemma 1";
let new_container_id: u32;

if init_container(new_container_id) == 0u8 {
    let result = false;
    for c in 0..32 {
        result = result || containers[*bv(c, 64l)].id==new_container_id;
    }
    *assert(result);
}
```

Again, the complete specification in Rust of both (1) and (2) is available in the appendix (A.3.3).

1.4 Scope of the project

Serval proposes different kinds of properties verifiable using the framework. While this project focuses on lemmas with a simple structure (a set of assumptions, some operations, and a set of assertions, arbitrarily combined), it is possible to extend it to support a broader set of Serval-defined constructs and ideas. For example, Serval proposes to verify the absence of undefined/unwanted behavior exploiting compiler capabilities and inserting a `(bug-on)` Racket call in the branches that should not be reached. If during symbolic execution `(bug-on)` is hit, then the verification fails. 4.1 will propose an idea to implement this differently, on top of the current work.

The aim of this project is not to add functionalities to Serval or to the verification environment, but rather to improve its usability by integrating it into Rust. As already introduced in the previous part (1.3) and detailed in the rest of the document, the Serval approach requires great effort, from the developer's side, to have a piece of code verified. Merging properties and code in the same project structure allows more concise and tight mapping between the software and its properties, making the verification of a complex hypervisor easier and more sustainable in the long term. Keeping the proof consistent with respect to code modifications is also straightforward.

It follows three phases. In the first moment, we try to understand how Serval can be used. After that point, we assess and experiment with the possibility to use it with Rust. In the end, thanks to all the gathered knowledge, we propose and implement a new language that is mapped to Racked code to express verifiable properties.

2 How to use Serval?

In the second section, there will be a description of Serval and the verification workflow to be used to verify simple programs. In particular, this workflow will be ported and integrated into Rust, as explained in the third section. The aim of this section is not to provide a step-by-step guide about how to prove code with Serval, but rather to give direction about it, as an addendum to the provided examples and code snippets.

2.1 Components

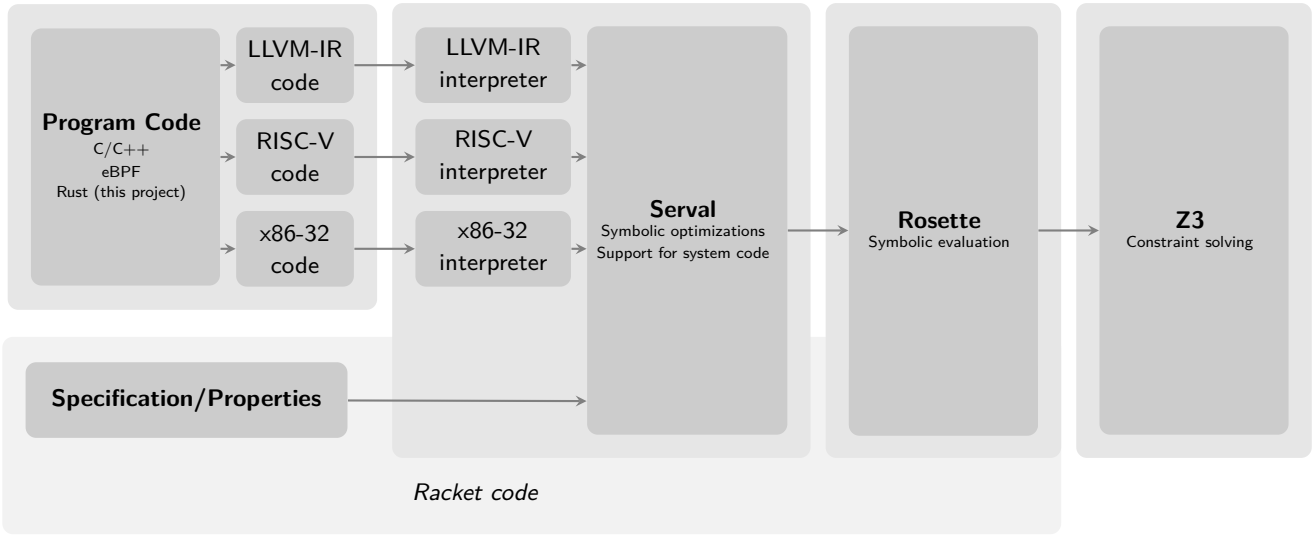
Serval [1] is a Racket library, and it is based on the following tools and programs.

Racket [5] is an extensible programming language, with a syntax similar to Lisp. It provides support for metaprogramming and parsing; Serval exploits these functionalities to build interpreters.

Rosette [6] extends Racket with constructs for symbolic analysis and verification of programs. Without being exhaustive, this language provides symbolic variables and lifts Racket constructs (e.g. `if-else`, expressions...) to support them. The code in lemmas is written and verified mainly using Rosette.

Z3 [7] is a theorem prover by Microsoft Research, on the top of which Rosette provides symbolic functionality. It is widely used in the industry and by other verification frameworks. For instance, also Prusti uses it.

The schema below represents how the components are interconnected.



2.2 Assumptions and Limitations

Due to the type of symbolic evaluation used, Serval requires the programs to be finite. A common definition of this property is that all possible execution traces must be finite in the number of steps. As an example, loops must always be bounded. Unbounded loops without specific constraints could make the set of possible values associated with variables increase arbitrarily, together with the feasible code paths that the execution may follow.

In addition to all the assumptions listed in the Serval paper (section 3.5 of [1]), the entire parser and translator to Racket proposed in this paper is not verified, thus it has to be assumed correct. Further, although Serval proposes a way to reduce the trusted code base for RISC-V programs, removing from it the assembler and linker, it is not possible to follow the same approach with LLVM-IR programs. Then, it is needed to trust the entire toolchain, the entire Serval framework and underlying tools, and this project.

Since Rosette does not support floating point bit vectors, the verified programs and properties cannot contain any non-integer math.

2.3 Overview of the verification workflow

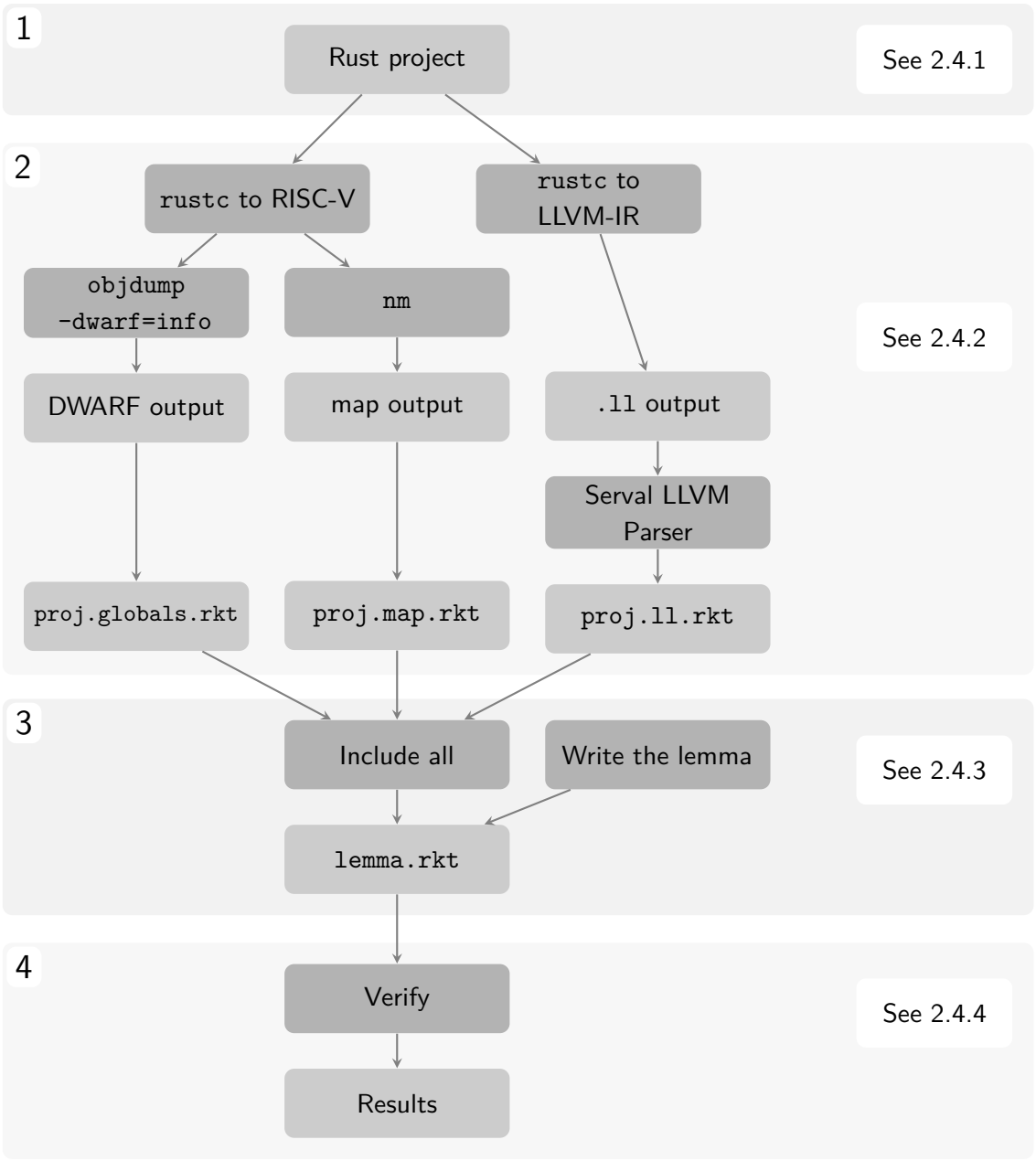
For the scope of the project, we use LLVM-IR as the language that is verified. We preferred this option, compared to using, for example, x86, for different reasons. In particular, Serval only supports a small part of the instruction set, making it difficult to add all the needed instructions for a full program to work. The focus of Serval regarding x86 support, indeed, is to be able to prove eBPF compiled rules. Furthermore, LLVM-IR abstracts the code representation from the underlying system, and it works seamlessly with any hardware architecture. On the other side, it lacks support, in case of usage, of specific machine instructions, which is not uncommon during the development of hypervisors and security monitors.

The source code has to be compiled twice. The first time, LLVM-IR code is generated, to be mapped to Rosette constructs. The second compilation is needed to obtain memory structure information. After the compilation and conversion, the resulting Racket files can be imported by the developer where the lemmas to be proved are defined. In the end, Serval provides a `Makefile` that has to be used to verify them, in a "unit test" fashion.

The next paragraph will describe the procedure, from the developer's side, to have a piece of code verified. Then, the subsequent paragraph will highlight some technical details about what happens. In the end, we list various issues encountered during the installation and usage of Serval, along with some proposed solutions for them.

2.4 Serval usage

The steps described in this part are all inferred by analyzing some Serval tests. In particular, it comes with a suite that checks its behavior with respect to the LLVM-IR language. Reusing that, it is possible to have a piece of software compiled to LLVM-IR first mapped to Racket code, then verified with respect to defined lemmas. The figure below represents the required steps to have a working proof. Steps 1 and 3 require manual effort, while 2 and 4 can be automated.



For a C/C++ project, the steps are very similar, the main difference is that `clang` is used as

compiler instead of `rustc`.

2.4.1 Prepare the Rust project

The Rust project to be verified must be without the standard library. It requires some further adjustments: a panic handler has to be defined. An empty project is provided, with all of the needed settings already put in place.

It is also required that the RISC-V Rust toolchain is installed. The rationale behind that is described in 2.4.2. For a guide on how to install it for cross compilation in an x86-64 environment, please see [8].

2.4.2 Compile it and obtain required Racket files

Two compilations are needed. In the appendix, we provide a script that does that automatically.

Verifiable code For that, the project is compiled into LLVM-IR intermediate language. Then, using a Serval-provided script, it is translated to Racket code. This kind of translation is straightforward for Serval. It considers only the code section of the file, and the LLVM-IR instructions [9] are directly mapped to Racket functions.

They are implemented by Serval, for example, to update bit vectors and to do calculations. Anyway, it does not implement the whole instruction set, but only a part of it. Nevertheless, they are sufficient to work with real Rust programs.

Instead, if we have to verify assembly programs instead, the approach is to parse directly the `objdump` output, instruction per instruction.

Memory structure To allow Serval to infer memory structure (in particular, global variables), the code has to be compiled to an actual executable/library, obtaining a `.so` file. Then, `nm` and `objdump` commands are applied to it to obtain respectively a symbols table `.map` and memory structure and debug information file `.globals`. At verification time, `.map` and `.globals` are read and interpreted, to generate, recursively, the memory definitions. Ideally, it is represented as nested dictionaries of bit vectors.

Given that existing tests use the RISC-V compiler, to reduce as much as possible potential inconsistencies and issues, this step has to be done with the RISC-V Rust toolchain. The file structure should be very similar if compiled to an x86 target, thus avoiding the need to have a different toolchain installed, but this was not tested during the semester. Potentially, there could be differences in the Dwarf file structure, regarding how variables are mapped to memory.

2.4.3 Write a lemma

Examples of simple lemmas are provided in the appendix. Lemmas could be written using Rosette constructs, please refer to the Rosette guide [6] to have a detailed overview of them. Serval, in this scope, is used mainly for two reasons. The first is that it provides a testing framework and it creates a working symbolic environment. Secondly, it maps Rust code to Racket.

The most important details, useful to write a working proof, are:

Rust-defined functions can be referred using a `@` as a prefix. They are defined in the `code` generated file, that has to be included. For example, if the `compute(a: i32, b: i32)` function is declared in the Rust source file, it can be called as `(@compute a b)` in the Racket file, where `a` and `b` are properly defined variables.

Global variables defined in Rust can be referred using a relatively verbose syntax. Let's assume that there is a global array `arr` of structs, each with a field `x`. Then, to obtain a reference to that value of the third element, we have, firstly, to get a reference to the array (`(llvm:symbol->block 'containers)`). Then, we can access to it using `(mblock-iload (llvm:symbol->block 'containers) (list (bv 3 i64) 'x))`. The `list` operators contain the sequence of indexes/labels to access the specific item.

Variables It is possible to declare both symbolic and concrete variables. Although lemmas are just Racket code, usually in this context high importance is given to bitvectors. Serval already defines some shortcuts for common types (e.g. `i8`, `i32`...). For example, to declare the variable `a` as a 32-bit integer of value 8, it is sufficient to write `(define a (bv 8 i32))`. On the other hand, to declare a symbolic 64-bit variable `x`, one could write `(define-symbolic x i64)`. Note that, even if the variables are initialized in Rust code, during symbolic evaluation they are all symbolic values, with no assumption about them.

Assumptions and assertions The two statements `(assume cond)` and `(assert cond)` are used to declare preconditions and postconditions. `cond` is a boolean expression on symbolic and concrete values.

Integration in the testing framework A lemma should be added to the list of the executed tests, that is a declaration at the end of the file.

2.4.4 Verify it

As already explained in the overview, Serval provides a `Makefile` that automatically executes every test in the working directory. Thus, it is sufficient to run the `make` command to prove all existing lemmas.

2.5 Issues encountered

During this part of the project, we encountered some compatibility issues between compiled Rust and Serval.

At first, to have predictable function and variable names, it is needed to avoid Rust "mangling", which adds a random suffix to every definition at compile time to avoid name clashing. This could be obtained by annotating with `#![no_mangle]` every function that has to be referenced from the specification.

Even if disabling `std` library reduces the number of boilerplate code in the compiled output, there are still some function calls that are not exported properly to LLVM-IR code by LLVM. It requires that they have to be defined in a Racket file and imported. Although this is not fully predictable, since we only noticed one call (`panic_bounds_check`) that is regularly repeated, we defined a stub Racket function with the same name that does nothing.

There exist some Rust constructs that generate code with unsupported LLVM-IR calls. For example, using a range loop inside the body of a function leads to the insertion of `resume` instructions. This specific one is related to exception handling and has a complex behavior that cannot be simplified to an empty body, but it should be implemented along with the other exception-related instructions. Given that, range loops should be avoided in program code. On the other side, there are also simple LLVM-IR statements (like `llvm.expect`) that are used as optimization annotations and their effect is neglectable during execution; we declared them as empty functions.

We also noticed a couple of issues in the `DWARF` parser regarding the recursive traversal of the file, that required some modifications in the Serval library.

3 Integration in Rust

In the third section, details about the integration and the capabilities of the proposed language will be described. This also serves as documentation for the language. Primarily, it has as its goal to replace parts 2.4.2 and 2.4.3 of the verification workflow.

The code is available on GitHub, https://github.com/epfl-dcsl/serval-rust.

3.1 Overview and high-level description

The main aim of the language is to be able to express a wide variety of properties without sacrificing readability and writeability. For that scope, a subset of the Rust language has been mapped to Racket constructs. The two main statements that can be used, as procedural macros in the Rust language, are `#[verify]` and `#[define]`.

`#[verify]` has as a main aim the definition of a lemma. A piece of code written in this macro is mapped to a verifiable lemma in Racket. It is automatically embedded into the testing framework, ready to be verified using the appropriate command.

`#[define]` is used to define a Racket function that is not directly embedded into the testing framework. Its scope is to be reused in different parts of the lemmas or in other defined functions. This allows the developer to write code that is better structured and without redundancies.

Both of the statements require a defined identifier to represent their name. `![verify]` does not require a defined identifier if it is used before a Rust function, as it takes its name, but it is up to the developer to avoid duplicate definitions. In particular, two definitions with the same name will collide during the actual verification.

3.2 Syntax

The main objective of the proposed syntax is to be as similar as possible to a Rust program. Unluckily, there are some limitations, given by the structure of Rosette in particular, that requires some tricks to be considered. Many of them will be listed in the next paragraphs.

3.2.1 Lemma and function declarations

As already stated in 3.1, there are two kinds of annotations that are defined to be used inside the Rust program to provide verification functionality. Their syntax is almost the same. If we want to define a verifiable lemma, then we should write in a Rust file inside the project, at

the same level as the functions defined in Rust.

```
#[verify({
    "lemma_name";
    // lemma to be verified
})]
fn test_function() -> i32 {
    // function code
}
```

We can omit the identifier for `verify` lemmas. Their identifier will be the name of the Rust function that they are annotating. Regard that, in the case of two different lemmas that annotate the same function and do not have a name, they will collide and the resulting code will fail to verify.

It is possible to declare arguments for `define`, calling the `args` just after the function name.

```
#[define({
    "function_name";
    args(x, y, z);
    // function code...
})]
```

3.2.2 Variables

It is possible to define both concrete and symbolic variables. In general, concrete variables have a value assigned from their instantiation, while symbolic ones are defined just by their type (bitvector size, in particular). There is no difference in the way they are used and referred to. Rosette provides support for symbolic variables in almost all contexts that are considered in the scope of this work. For example, but not limited to, they can be used in expressions, function calls, conditional constructs, and so on.

Serval maps all variables and memory regions to bitvectors. They can, therefore, represent all variables of a Rust program, if defined of the correct size.

In any expression, numeric values must be suffixed with their size in bits. For example, when we want to define an `int` of 4 bytes with value 3, we should write `3u32`. Obviously, we can also use the signed alternative, like `3i32`. In Rosette, they are mapped to the same kind of bitvector, but the operations that are applied should be different.

An example:

```
let x = 15u8; // Define a concrete unsigned 8-bit variable with value of 15
```

```
let y : i64;  // Define a symbolic signed 64-bit variable
```

It is not possible to specify both a size (using the type annotation to the variables) and an initialization value. Variables with type annotation define symbolic variables, initialization values define concrete variables.

Variables can also be reassigned. Since variable names are not bound to types, there is no problem assigning a different kind of value to a variable.

```
let x = 15u8;
let y : i64;

y = x; // Now y has 15 as value
```

In case of need, the cast operator `as` can be used to resize (actually, to extend) a bitvector to a bigger size.

```
let y : i32;

let z = y as i64;
```

3.2.3 Expressions

Almost all binary operators available in Rust are supported and can be used to write arbitrarily complex expressions. As of now, since there is no elegant way to differentiate signed and unsigned operators, by default all of them are unsigned.

To overcome this limitation, two pseudo-functions `signed` and `unsigned` are provided. The expression inside of them is always treated as signed or unsigned, thus operations defined inside of them are mapped to the correct Rosette expression. Moreover, they can be arbitrarily nested to allow precise definitions of statements.

Unsigned binary operators, that behave the same in `signed` and `unsigned` contexts, are:

- Math operators: `+`, `-`, `*`
- Boolean operators: `&&`, `||`
- Bitwise operators: `^`, `&`, `|`
- Shift operator: `«`, the two operands must be bitvectors of the same size
- Equality: `==`

Signed operators, that may produce a different result if used in `signed` or `unsigned` context, are:

- Math operators: `/`, `%`

- Shift operator: », in a signed context, leading one is extended
- Comparison operators: <, <=, >, >=

```
let z = 3u8 - 5u8; // Result is 0xFE

// u is false, because z is considered as unsigned
let u = z < 0u8;          // 0xFE > 0x00
// s is true, because the comparison is now inside of a signed context
let s = signed(z < 0u8); // 0xFE => -0x02 < 0x00
```

3.2.4 Function calls

In the current environment, there are two kinds of function calls.

Rust defined functions are the ones that are defined in the code as normal functions. In general, they should contain the code that has to be verified. They should be written as normal function calls, for example:
`let result = function_call_test(a, b, 3u8);`

Racket defined functions usually are already defined functions in the Racket/Rosette/Serval environment. They could also be other blocks of code defined with the `define` macro. Their calls should be prepended with an asterisk. For example, if we want to print a variable for debugging purposes:

```
*print("Value of a is: ");
*println(a);
```

In general, this differentiation must be enforced and respected. As described previously, "code" functions are called differently with respect to "racket" functions.

It is important to remember the simple syntax of Racket. Almost all constructs are "function-like" statements, thus a wise usage of this described parser could be used also to form expressions that were not considered in this report.

3.2.5 Memory access

The parser support memory access to *global* variables. They can be arbitrarily complex nested arrays and structs. The syntax is the standard for a Rust program. For example, accessing the field `f` of the 4th element of array `arr` should be written as:

```
let val = arr[3u64].f;
```

Indexes for array accesses could be arbitrarily complex expressions, but the resulting value must be a 64-bit unsigned bitvector. Particular attention has to be used in the case of `for`

loop indexes, see 3.2.7 for details.

As of now, it is not possible to refer to global variables that are not arrays or structs.

3.2.6 Conditional execution

`if` and `else` statements can be written as usual Rust statements. Obviously, their condition should be a boolean (symbolic or concrete) value.

3.2.7 `for` loops

Only ranged `for` loops can be written. Further, they always start from 0, that could also be not written explicitly, and end with an explicit integer value/ No variable or expression could be written as a range end, only numbers. A simple example is:

```
for i in 0..8 {....}
```

It is important to remember that variable `i`, inside of the scope of the loop, is not a bitvector but an actual integer, as defined in the Racket language. Thus, it must be converted to a bitvector in case of usage with Serval/Rosette constructs. Namely, if we want to access the `i`th element of a vector `vec`, we should use the `bv` function to have it converted to a 64-bit bitvector.

```
for i in 0..8 {
    *println(vec[bv(i, 64)]);
}
```

3.2.8 Arbitrary Racket code

Since it is not possible to map any possible Racket construct to Rust, it is also allowed to use directly raw Racket code, using the `raw` pseudo-function. For example, if we want to call the `println` function directly, we write:

```
raw("(println \"test\")");
```

To be precise, `println`, as any other Racket function, can be called also using Rust syntax, see 3.2.4. It is obviously possible to refer to defined Rust variables and other functions inside of `raw`, and vice versa. Names are mapped as they are between Rust and Racket.

3.3 Implementation

The translation to Racket code happens in two steps. In the first, each macro statement is parsed independently and its contents are put in a JSON file as a syntax tree. Then, all the resulting files are collected and expanded into Racket files. The compilation of the actual Rust code remains unchanged and output files are mapped by Serval as already explained in 2.4.2.

There are no semantic and type checking enforced. All of these are delayed to verification time. Racket will try to execute the output file and, in case of errors, they will be reported. Even if this is a simplification in the translator, it allows more freedom for the developer to express even code that was not "planned" during the project. Further, it is complex to check, for example, if a name exists in a relatively complex environment created by the intersection between Racket, Rust, and various libraries.

3.3.1 First step: Rust code parsing

Parsing is done by Syn, a Rust library specifically targeted to parse Rust code in the context of macros. The result is a syntax tree formed by nested structures and enumerations.

All the required logic is implemented in the module `macroslib`, which should be imported into the code to be verified.

This result is then unwinded using recursive calls. Only the subset of the syntax of interest is interpreted. In case of an unexpected token, an error is raised.

Errors are displayed using the Rust compiler library. It allows to highlight precisely what, and where, is the problem, and some IDEs support them allowing even better spotting. This has required some additional effort since this feature is supported only by the Nighly toolchain, but that adds some constructs in the LLVM-IR output that are not supported by Serval. As a workaround, one compilation pass is done with the Nightly toolchain, and the other with the Stable.

3.3.2 Second step: Racket code generation

The logic of this second part is included in another module, which is a stand-alone Rust executable program.

All the JSON files are collected. Then, based on the syntax tree, Racket code is recursively generated. All the lemmas and functions are put in a single Racket file, that can be verified using, for example, the `make` command in the output directory.

3.4 Evaluation and limitations

We experimented with the proposed language with two objectives of correctness in mind. The first, straightforward, is that generated Racket code should be valid and meaningful. The second is that a valid Racket lemma should be writable in Rust and should have the same interpretation and the same result. For testing purposes, we wrote three classes of Rust functions, with increasing complexity: simple operations on variables, operations on arrays, and operations on arrays of structs with functions that resemble the ones that may be implemented in a security monitor. An extract of the former two is available in A.1 and A.2, while an example of the latter is reported in A.3. Then, we wrote some lemmas in Racket and their equivalent in Rust. Our tests show that their behavior is the same, specifically, the same properties are proved true or wrong, with the same counterexamples. The direct mapping between Rust and Racket names makes the comparison straightforward.

There are some sugared syntax constructs in Racket, for example, the single quote (`'`) operator, that have still no Rust equivalent. However, this is just a short-hand alternative for `(quote ...)` expression, that can be defined as a function call (`quote(...)`) in Rust. Similarly, it is also not possible to declare and use lists/arrays with the Rust syntax yet, even if this could be implemented, but only with a function call. An integer array `[1, 2]` may be declared as `list(1u32, 2u32)`. Though, this requires some knowledge of Racket and Rosette.

The main limitation of this approach is that it is not possible to express top-level constructs in a Racket file that are different from `(define (name) (body...))`. This means that the developer cannot, for example, import additional source files, or export items outside of the scope of the current file. Nevertheless, in the considered context these features are not strictly necessary, and that may be solved by adding a new kind of macro or, if a file has to be always included, it may be statically added to the generated Racket template. Further, Racket supports, in name identifiers, more symbols compared to Rust, e.g. the hyphen `-`, that cannot be expressed in the current implementation.

4 Conclusions

The proposed language achieves the initial objective of having Serval work with Rust. In particular, it is now possible to verify Rust programs using Serval and write lemmas directly in the Rust code. It is also straightforward to add new constructs to the parser in case of need. The two-step structure of the parser also decouples the Rust lemmas from the Racket code, making it feasible to partially reuse them with different languages and provers.

The advantages are varied, but the most important is regarding developer experience. Previously, it was needed to write code in two different languages, having to orchestrate two completely disjoint file structures to have even a simple piece of code verified. Now, it is sufficient to write all the code in a single project and have it proven with a couple of commands.

There are, however, some aspects that could be improved. Compared to a standard Rust program, the proposed language adds some verbosity, especially regarding literal numbers definition. Not to mention the fact that it lacks type checking, delaying the discovery of potential errors to verification time.

Also, performance is an issue that has not been taken into account in the scope of the project. For symbolic evaluation, the shape of a program is determinant to avoid state explosion or proofs that do not terminate in a reasonable amount of time. Although Serval proposes some optimizations to reduce this problem, it was easy, during the course of the project, to clash with simple programs of which the evaluation runs indefinitely. Since programs in this context have to be finite, however, the verification always terminates, but it may take too much time.

Using the latest versions of Rust nightly, we noticed that a new capability of the language, *opaque pointers* that do not have a pointee type associated, has been introduced to replace standard pointer definitions. Even if this feature can be disabled with a compiler flag in the transition period, it is expected that it will become the new standard. Serval interpreter does not support this new construct and, thus, it should be eventually updated.

4.1 Further ideas

There is a lot of room for further improvements before claiming to have a complete verification environment in Rust.

As of now, it is possible to declare and verify simple properties using the proposed language. Serval, however, lists different kinds of properties that can be verified for a program and that could require instrumenting the code. For example, the existence of undefined/unwanted behavior is one of the easiest to implement on top of the current work.

A straightforward solution is to use a Rust boolean global variable `undefined` and a function to update it, which is called in the circumstance of unwanted behavior. Then, using an `assertion`, the value of `undefined` is checked to be `false`. All of these statements can be desugared from more elegant constructs written in Rust lemmas or code, as macros. It is also possible to insert `(assume false)` in specific positions of the program, but this may require modifying Serval and the Racket files that it generates.

Another aspect to consider is that each lemma and function call is considered independently from the others. An optimization could consist in reusing already verified lemmas, or executed pieces of programs, in other contexts when needed. For example, when the same function call is reached with the same (or weaker) assumptions on the input values, the result should be the same, or, in general, should contain the specific expected result. An approach that has been proposed to solve this kind of problem is Green [10].

References

[1] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 225–242, New York, NY, USA, 2019. Association for Computing Machinery.

[2] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers. The prusti project: Formal verification for rust (invited). In *NASA Formal Methods (14th International Symposium)*, pages 88–108. Springer, 2022.

[3] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), jul 1976.

[4] Examples of security monitors verified by serval, https://github.com/uw-unsat/serval-sosp19/tree/master/monitors.

[5] Racket language, https://racket-lang.org/.

[6] The rosette guide, https://docs.racket-lang.org/rosette-guide/index.html.

[7] Z3 theorem prover, https://github.com/Z3Prover/z3.

[8] Daniel Mangun. Rust cross compilation, https://danielmangum.com/posts/risc-v-bytes-rust-cross-compilation/, jan 2022.

[9] Llvm-ir reference, https://llvm.org/docs/LangRef.html.

[10] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

A Appendix

A.1 Properties expressed in Racket

To obtain the files needed by Serval, the following script may be useful. It should be run from the Rust project top-level folder.

```
# Output path for the .rkt files
BASE_PATH=/home/filippo/serval/rust/gen

# Compile to LLVM-IR and map to Racket constructs
cargo rustc --release -- --emit=llvm-ir -C panic=abort
racket /home/filippo/serval/serval/bin/serval-llvm.rkt --
 ↪  target/riscv64gc-unknown-linux-gnu/release/deps/rusty_risc.ll >
 ↪  $BASE_PATH/sumrust.code.rkt

# Compile to RISC-V to obtain memory structure
cargo rustc --release -- -C panic=abort

echo "#lang reader serval/lang/nm" > $BASE_PATH/sumrust.map.rkt
riscv64-linux-gnu-nm --print-size --numeric-sort
 ↪  "target/riscv64gc-unknown-linux-gnu/release/librusty_risc.so" >>
 ↪  $BASE_PATH/sumrust.map.rkt

echo "#lang reader serval/lang/dwarf" > $BASE_PATH/sumrust.globals.rkt
riscv64-linux-gnu-objdump --dwarf=info
 ↪  "target/riscv64gc-unknown-linux-gnu/release/librusty_risc.so" >>
 ↪  $BASE_PATH/sumrust.globals.rkt
```

Given some simple Rust functions

```
#[no_mangle]
fn sum(x: u32, y: u32) -> u32{
  let r: u32 = x + y;
  return r;
}

#[no_mangle]
fn fake_equality(x: u32, y: u32) -> u8{
  if x==y || (x==0&&y==1) { return 1; }
  else { return 0; }
}
```

```
#[no_mangle]
static array_to_sum: [u32; 5] = [1,2,3,4,5];

#[no_mangle]
unsafe fn sum_array() -> u64{
    let mut result: u64 = 0;
    let mut i: usize = 0;

    while i<5 {
        result += array_to_sum[i] as u64;
        i = i + 1;
    }

    return result;
}
```

we can define some properties over them

```
(define (call-sum)
    (define-symbolic a i32)
    (define-symbolic b i32)
    (define r (@sum a b))
    ; We prove both that sum function does an addition and that is commutative
    (assert (bveq r (bvadd b a)))
)

(define (call-equality)
    (define-symbolic x i32)
    (define-symbolic y i32)
    (define result (@fake_equality x y ))
    ; This property fails to verify because fake_equality is not properly
    ↪ defined
    (cond
        [(bveq x y) (assert (bveq result (bv 1 i8)))]
        [else (assert (bveq result (bv 0 i8)))]
    )
)

(define (call-array-sum)
    ; This function assign values 1, 2, 3, 4, 5 to the array
    (@init_array)

    (define expected_result (bv 0 i64))
    (define b0 (llvm:symbol->block 'array_to_sum))
    (for ([i 5])
      (define curr (zero-extend (mblock-iload b0 (list (bv i i64))) i64))
      (set! expected_result (bvadd expected_result curr))
```

```
    )

    (assert (bveq @sum_array expected-result))
)
```

At the end, they should be integrated and hooked into the testing framework

```
(define rust-tests
    (test-suite+
       "Tests for rust functions"
       (parameterize ([llvm:current-machine (llvm:make-machine symbols
        ↪ globals)])
           (test-case+ "sum" (check-function0 call-sum))
           (test-case+ "fake equality" (check-function0 call-equality))
           (test-case+ "array sum" (check-function0 call-array-sum))
       )
    )
)

(module+ test
  (time (run-tests rust-tests)))
```

A.2 Properties expressed in Rust

We can express exactly the same properties directly in Rust. There is no need to write code in Racket or integrate them into the testing framework, all is done automatically behind the scenes.

```
#[verify({
  let a: i32;
  let b: i32;
  *assert(sum(a, b) == b+a));
})]
#[no_mangle]
fn sum(x: u32, y: u32) -> u32{
  let r: u32 = x + y;
  return r;
}

#[verify({
  let x: i32; let y: i32;

  let result = fake_equality(x, y);
  if x == y { *assert(result==1u8); }
  else { *assert(result==0u8); }
```

```
})]
#[no_mangle]
fn fake_equality(x: u32, y: u32) -> u8{
  if x==y || (x==0&&y==1) { return 1; }
  else { return 0; }
}

#[no_mangle]
static array_to_sum: [u32; 5] = [1,2,3,4,5];

#[verify({
    init_array();

    let expected_result = 0u64;
    for i in 0..4 {
        expected_result = expected_result + array_to_sum[*bv(i, 64l)];
    }

    *assert(sum_array() == (expected_result as u64));
})]
#[no_mangle]
unsafe fn sum_array() -> u64{
    let mut result: u64 = 0;
    let mut i: usize = 0;

    while i<5 {
        result += array_to_sum[i] as u64;
        i = i + 1;
    }

    return result;
}
```

A.3 Motivating Example

A.3.1 C sample code

The code that we want to verify as example is

```
typedef struct {
    int id;
    // ... other fields
} container_t;

container_t containers[32];
```

```
int container_exists(int container_id){
    for(size_t i = 0; i<32; i++){
        if(containers[i].id == container_id) return 1;
    }
    return 0;
}

int init_container(int container_id){
    if(!container_exists(container_id)){
        for(size_t i = 0; i<32; i++){
            if(containers[i].id == 0){
                containers[i].id = container_id;
                return 0;
            }
        }
        return 2;
    } else return 1;
}
```

A.3.2 Racket Lemmas

```
#lang rosette

(require (except-in rackunit fail) rackunit/text-ui rosette/lib/roseunit
 ↪  (prefix-in llvm: serval/llvm) serval/lib/unittest serval/lib/core)
(require "gen/code.map.rkt" "gen/code.globals.rkt" "gen/code.code.rkt")

(define (lemma-1-prove-creation)
  (define-symbolic new_container_id i64)

  (if (bveq (@init_container new_container_id) (bv 0 i8))
    (let ()
      (define contains #f)
      (for ([c 32])
        (define current_id (mblock-iload (llvm:symbol->block 'containers) (list
         ↪  c 'id)))
        (set! contains (or (bveq new_container_id current_id) contains))
      )
      (assert contains)
    )
    (assert #t)
  )
)

(define (lemma-2-prove-fail-on-existing)
  (define-symbolic new_container_id i32)
```

```
  (if (bveq (@container_exists new_container_id) (bv 1 i8))
    (assert (bveq (@init_container new_container_id) (bv 1 i8)))
    (assert #t)
  )
)

(define (check-function0 f )
  (define r (verify (f)))
  (assert (unsat? r) r)
)
(define rust-tests
  (test-suite+
   "Tests"
   (parameterize ([llvm:current-machine (llvm:make-machine symbols globals)])
   (test-case+ "lemma 1" (check-function0 lemma-1-prove-creation))
   (test-case+ "lemma 2" (check-function0 lemma-2-prove-fail-on-existing))
   )))
(module+ test
  (time (run-tests rust-tests)))
```

A.3.3 Rust Lemmas

The same properties can be expressed as Rust macros, leaving out actual Rust program code, as:

```
#[verify({
    "lemma 1";
    let new_container_id: u32;

    if init_container(new_container_id) == 0u8 {
        let result = false;
        for c in 0..32 {
            result = result || containers[*bv(c, 641)].id==new_container_id;
        }
        *assert(result);
    }
})]
#[verify({
    "lemma 2";
    let new_container_id: u32;

    if container_exists(new_container_id) == 1u8 {
        *assert(init_container(new_container_id)==1u8);
    }
})]
```