ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WECPR02



# BENEFITS AND DRAWBACKS OF USING RUST IN AN EXISITNG C/C++ CODEBASE*

B. S. Martins†[1], Facility for Rare Isotope Beams, East Lansing, USA
[1]also at Columbia University, New York, USA

*Abstract*

Mozilla has recently released a new programming language, Rust, as a safer and more modern alternative to C++. This work explores the benefits (chiefly the features provided by Rust) and drawbacks (the difficulty in integrating with a C ABI) of using Rust in an existing codebase, the EPICS framework, as a replacement for C/C++ in some of EPICS' modules.

## INTRODUCTION

Mozilla's new systems programming language, Rust, promises to make entire classes of bugs detectable and preventable at compile time [1]. Its first stable version, dubbed 1.0, was released on May 2015. Rust has a heavy focus on memory safety and uses novel concepts, such as resource lifetimes and the borrow checker, to achieve that goal. These features, however, come at the cost of increased language complexity. Rust also aims to be fast and provide binary compatibility with C while providing high level constructs, making it a great candidate for replacing both C and C++ as system languages. While Rust does not have full feature parity with C yet [2], the language is rapidly evolving in this direction.

EPICS [3] is an industrial controls system framework used in several big science facilities around the world. EPICS is primarily written in C and C++ and has been incorporating contributions from several people along its three decades of existence. Therefore, its codebase contains a mixture of legacy and modern C and C++ code that presents several opportunities for improvement.

Rust's safety claims and modern features are enticing for projects like EPICS. This work investigates the use of the Rust language in the context the EPICS framework in an attempt to answer the following research questions:

1. Would Rust have prevented actual EPICS bugs?
2. Is it straightforward to translate C/C++ code into Rust?
3. Is it feasible to rewrite parts of EPICS into Rust?
4. Is it worth it to rewrite a big C/C++ project into Rust?

Question 1 will be answered by first examining EPICS' issue tracker [4] and classifying its bugs into a few categories, and then rewriting a few representative bugs in Rust to verify if its compiler would have caught them. Questions 2, 3 and 4 will be answered by evaluating the manual reimplementation of a single EPICS base component, iocsh, into Rust.

## RUST'S MEMORY SAFETY FEATURES

While Rust has many modern programming language features, such as first-class functions, closures, algebraic types, async/await, etc., its strength lies in its ownership system, which can be divided into three concepts: **ownership**, **borrowing** and **lifetimes**. These concepts play a fundamental role in ensuring memory safety.

### *Ownership*

Rust's ownership mechanism ensures, at compile time, that all values in a Rust program have an owner. Typically, the owner of a value is the variable the value was first assigned to. When that first value is assigned to a second variable, it is said that the value is *moved*, and the first variable loses *ownership* to the second variable. After a value is *moved*, the first variable cannot be used anymore to reference it. For example, in Listing 1, the variable `a` is the first *owner* of the vector containing the values 1, 2 and 3. Then, on the following line, the vector is *moved* to the variable `b`, which means `a` no longer *owns* the vector; trying to access `a` again would violate Rust's *ownership* constraints, so the compiler prevents it from happening.

Listing 1: Ownership example

```
1    fn main() {
2      let a = vec![1,2,3];
3      let b = a;
4      println!("{:?}", a);
5    }
```

As shown in Listing 2, the compiler emits helpful messages: it tells where the value was first moved (at line 3 when being assigned to `b`) and hints that the particular type does not implement the `Copy` trait. The Copy trait indicates that, rather than being *moved*, the value can be instead *copied* to the destination. All primitive types implement the `Copy` trait. If non-`Copy` values had to be moved back and forth between owner variables Rust would be a very impractical language. Therefore, there is a mechanism for taking references to a value, which is called borrowing.

* This material is based upon work supported by the U.S. Department of Energy Office of Science under Cooperative Agreement DE-SC0000661, the State of Michigan and Michigan State University.
† martins@frib.msu.edu
[1] Category of interest for this study

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WECPR02

Listing 2: Ownership example compilation error

```
error[E0382]: borrow of moved value: `a`
 --> src/main.rs:4:22
  |
3 |   let b = a;
  |           - value moved here
4 |   println!("{:?}", a);
  |                    ^ value borrowed
here after move
  |
  = note: move occurs because `a` has
type `std::vec::Vec<i32>`, which does
not implement the `Copy` trait
```

### *Borrowing*

Borrows are similar to C++ references. However, while C++'s references are guaranteed to be non-null and are mutable by default, Rust's so-called borrow-checker ensures that the programmer can take as many *immutable* borrows as desired, while only one *mutable* borrow is allowed at a time. Thus this mechanism prevents data races at compile time, even across threads. A simple example of borrowing is shown in Listing 3.

Listing 3: Borrowing example

```
1    fn print_vec(v: &Vec<i32>) {
2      println!("{:?}", v);
3    }
4    fn main() {
5      let a = vec![1,2,3];
6      print_vec(&a);       // print_vec
"borrows" a
7      let b = a;           // a is
moved to b
8      println!("{:?}", b);
9    }
```

### *Lifetimes*

The third aspect of Rust's ownership system is lifetimes. Each value in Rust has a lifetime associated with it. Most of the time Rust can infer the lifetimes, but sometimes they need to be made explicit. Lifetimes are a way of giving names to scopes. Therefore, essentially, the compiler ensures that the references needed in different functions don't go out of scope, preventing use-after-free and similar bug classes.

## EPICS ISSUES ANALYSIS

In order to answer Question 1, an analysis of the EPICS' issue tracker was performed. The issues were first filtered by the tags "Fix Committed" or "Fix Release". Then, the most recent 185 issues (spanning almost 10 years), were manually classified, with the help of a custom tool, into one of 10 categories at the author's discretion. The results of the classification are summarized in Table 1.

Table 1: EPICS Issues Classification

| Issue Category | Count | Percent |
|---|---|---|
| Logic | 97 | 52.4 % |
| Build system | 36 | 19.5 % |
| Race condition[1] | 15 | 8.1 % |
| Buffer overflow[1] | 9 | 4.9 % |
| Improvement | 9 | 4.9 % |
| Use-after-free[1] | 7 | 3.8 % |
| Type cast[1] | 5 | 2.7 % |
| Null pointer dereference[1] | 4 | 2.2 % |
| Third-party | 2 | 1.1 % |
| Return from stack[1] | 1 | 0.5 % |
| Total | 185 | 100 % |

### *Issue Categories*

**Logic** Issues of this class are bugs on the software operation logic that don't cause the system to crash. These usually describe incorrect or unexpected behavior by the software and are not of interest for this study.

**Build system** Issues of this class are related to EPICS' custom build system and are not of interest for this study.

**Race condition** A number of issues arose from race conditions between EPICS many execution threads. These threads often communicate via shared memory, which can easily lead to issues if data access is not properly synchronized. Rust claims to be capable of detecting this kind of issue using their concept of lifetimes and ownership.

**Buffer overflow** Issues that referred to out-of-bounds buffer reads and/or writes were classified as buffer overflows. Rust detects invalid accesses at runtime and safely stops the program (which Rust calls "panic"), instead of potentially allowing the program to continue like C and C++ can.

**Improvement** Issues with this classification were not bugs at all; they were opened to request improvements or new features to be implemented. These issues are not of interest for this study.

**Use-after-free** Some issues referred to code that was attempting to use a resource after it had been released. Rust has the concept of lifetimes to prevent the use of a resource after it is "Dropped" (in Rust-speak).

**Type cast** A few issues arose from C/C++'s implicit type casting and from unaligned explicit casts on platforms that don't support unaligned access. Rust prevents this class of bugs by having a strict type system that requires explicit casting.

**Null pointer de-reference** Issues of this nature refer to code that tries to de-reference a pointer that was set to null. Rust can statically detect some cases of null dereference.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WECPR02



**Third-party** EPICS relies on a few third-party libraries, such as yacc and flex, to build parsers for its Domain Specific Language for specifying IOC database files, and this class of issues refers to bugs with the interaction between EPICS and such libraries. Issues of this nature are not of interest for this study.

**Return from stack** The single identified issue of this nature refers to a case where a function was allocating data on the stack and then passing that data to a second function to be run on a separate thread. Since the threads would run independently, there was no guarantee that the data seen by the second thread would remain valid. It is equivalent to a function simply returning a reference to data allocated on its stack. Rust can defend against this kind of bug with the concept of lifetimes.

## REWRITING ISSUES IN RUST

Of the 10 listed categories, 6 are of interest of this study: race condition, buffer overflow, use-after-free, type cast, null pointer de-reference and return from stack. One representative bug from each category was chosen to be rewritten in simplified C or C++ and in Rust. Then, each pair of programs was compiled by their respective compilers and run, if compilation succeeded. The difference in behavior between compilers and compiled programs was then analyzed. While all 6 representative bugs were rewritten, only 2 will be examined in this paper, in the interest of space: a use-after-free and a null pointer de-reference bug.

All tests were run on an Intel i7-4700MQ CPU with 16 GB of RAM, running Ubuntu 18.04.2. The C and C++ compilers used were the system's gcc and g++, respectively, both at version 7.3.0. The Rust compiler version was 1.33.0, the latest one available at time of testing.

### *Use-After-Free Bug*

One example of an use-after-free-bug was found in the EPICS issue #861214. The real bug is triggered when an EPICS IOC is exiting, and it would be too contrived to be reproduced here. An equivalent example is shown instead in Listing 3a, with a Rust translation shown in Listing 3b.

Listing 3a: Use-after-free in C++

```
1    #include <stdio.h>
2    struct Dummy {
3      int d;
4      Dummy(int d):d(d) {}
5    };
6
7    int main(void) {
8      Dummy *dummy = new Dummy(42);
9      delete dummy;
10     printf("%d\n", dummy->d);
11     return 0;
12   }
```

Listing 3b: Use-after-free in equivalent Rust

```
1    #[derive(Debug)]
2    struct Dummy(i32);
3
4    fn main() {
5      let a = Dummy(42);
6      drop(a);
7      println!("{:?}", a);
8    }
```

Listing 3c: Use-after-free in C++ compilation results

```
$ g++ -Wall -Wextra main.cpp -o main
$ ./main
0
```

Listing 3d: Use-after-free in Rust compilation results

```
$ cargo build
   Compiling bug861214 v0.1.0
(/home/bmartins/bug861214)
error[E0382]: borrow of moved value: `a`
 --> src/main.rs:7:22
  |
6 |   drop(a);
  |        - value moved here
7 |   println!("{:?}", a);
  |                    ^ value borrowed
here after move
  |
  = note: move occurs because `a` has
type `Dummy`, which does not implement
the `Copy` trait
```

As seen in Listings 3c and 3d, the results illustrate a stark contrast between the languages capabilities. The C++ code involved in this bug accesses a field of an object *after* it has been freed without objections from the compiler. g++, even with warnings enabled, doesn't raise an issue with the program. Rust, on the other hand, is able to catch the bug at compile time, by using its concept of ownership: the structure instance of `Dummy` is first owned by the variable `a`. Then, on the next line, it is *moved* to the function `drop`. Hence, after `drop` executes, `a` is not valid anymore, and cannot be accessed. This design choice is so fundamental that it makes for a clever implementation of the function `drop`: `drop` takes the value (by *move*) and then does nothing.

### *Null Pointer Dereference Bug*

Null references were once called a "billion dollar mistake" [5], given how dangerous and costly they can be. The EPICS issue #1369626 has such a bug: EPICS allows

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WECPR02

the user of its libraries to register callback functions on certain events, via function pointers. In this particular case, a null pointer was being inadvertently passed to the registration function, which happily accepted it. However, when the relevant event triggered, the trigger code tried to call the registered function via the function pointer, without checking that the pointer was actually valid, leading to a null pointer de-reference. A simplified example of such bug is shown in Listings 4a-4d.

Listing 4a: Null pointer deref. in C

```
1    typedef void caEventCallback-
Func(int arg);
2    int main(void) {
3      caEventCallbackFunc *cb = NULL;
4      cb(0);
5      return 0;
6    }
```

Listing 4b: Null pointer deref. in Rust

```
1    type CaEventCallbackFunc = *const
fn (i32) -> ();
2    fn main() {
3      let cb : CaEventCallbackFunc =
std::ptr::null();
4      (*cb)(0);
5    }
```

Listing 4c: Null pointer deref. in C compilation results

```
$ gcc -Wall -Wextra main.c -o main
$ ./main
Segmentation fault (core dumped)
```

Listing 4d: Null pointer deref. in Rust compilation results

```
$ cargo build
   Compiling bug1369626 v0.1.0
(/home/bmartins/comsw6156/pa-
per/rust/bug1369626)
error[E0133]: dereference of raw pointer
is unsafe and requires unsafe function
or block
 --> src/main.rs:4:5
  |
4 |    (*cb)(0);
  |    ^^^^^ dereference of raw pointer
  |
  = note: raw pointers may be NULL, dan-
gling or unaligned; they can violate
aliasing rules and cause data races: all
of these are undefined behavior
```

gcc allows this: from gcc's point of view, the programmer is ultimately responsible for their pointers, full stop. Rust, interestingly, doesn't disallow the use of null pointers *per se*; Rust's compiler does halt compilation if it finds a raw pointer being de-referenced, but the error message tells us that the de-reference *would* be allowed if it was done inside an unsafe block. The unsafe block is an escape hatch from the strictness of Rust's compiler: it is a way for the programmer to tell the compiler that they know what they are doing and that they performed the appropriate memory safety checks for that particular snippet. From Rust's perspective, gcc can be thought of as running in unsafe mode all the time!

### *Issue Reimplementation Summary*

As seen in Fig. 1, most of the 185 classified issues, 144 (77.8%), belonged to 4 categories that were not the target of this study, since they wouldn't, in principle, benefit from Rust's static analysis: logic bugs, build system, improvements (feature requests) and bugs in third-party libraries.

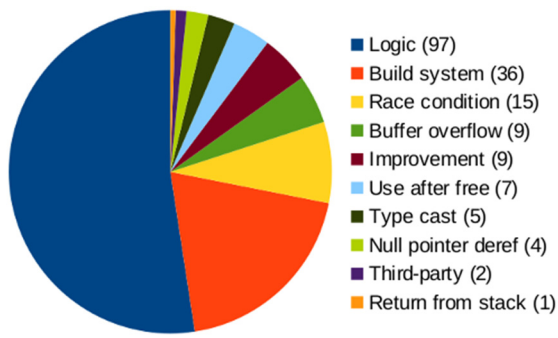


Figure 1: EPICS issue categories distribution.

The remaining 41 (22.2%) issues were further inspected with the aim of having one issue from each class to be selected for reimplementation in C or C++ and Rust.

Rust's compiler, using the default out of the box options, was able to remarkably catch all 6 tested bugs: 5 bugs were caught at compile time, by performing static analysis, and 1 at run time through bounds checking. All bugs caught at compile time prevented the compilation from proceeding. Most of the error messages emitted by Rust's compiler were informative, providing a way to get more information on the particular emitted error and, sometimes, a suggestion on what to do to fix the problem that it encountered. In light of these findings, were EPICS be written in Rust, it would not be unreasonable to extrapolate that the 32 (17%) bugs in these 5 classes wouldn't even have appeared in the issue tracker. It can also be argued that the 9 (4.8%) bugs from the sixth class (buffer overflow) would have been more easily found and fixed given the informative panic message issued by Rust at runtime, if run with the debug binary.

The GNU compilers, on the other hand, only caught 2 out of the 6 bugs, and in both cases they just emitted warnings instead of halting the compilation. It can be argued that the `-Werror` flag could have been passed to gcc and g++ in order to halt the compilation on warnings, but since their warnings can vary greatly by platform and

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WECPR02



compiler version, `-Werror` can potentially make the build system brittle. It is also interesting to note that 2 bugs that didn't even get a warning emitted by gcc (buffer overflow and null pointer de-reference) resulted in an outright crash at runtime.

Table 2: EPICS Issues Reimplementation Findings

| Bug class | Lang | Bug caught? | Crash? |
|---|---|---|---|
| Race condition | C++ | No | No |
| | Rust | At compilation | - |
| Buffer overflow | C | No | Yes |
| | Rust | At execution | No |
| Use-after-free | C++ | No | No, but wrong result |
| | Rust | At compilation | - |
| Type cast | C | No (warning) | No |
| | Rust | At compilation | - |
| Null pointer deref. | C | No | Yes |
| | Rust | At compilation | - |
| Return from stack | C | No (warning) | Yes |
| | Rust | At compilation | - |

Regarding Rust's usability, Rust was very straightforward to install and simple to start coding with. The Rust book [6] presents an excellent overview of the language, especially for people already familiar with other programming languages. The sample snippets of code in C and C++, albeit short, could be translated with little effort into Rust, resulting in an almost direct translation in all cases. Furthermore, the error messages given by the compiler were informative, giving precise bug locations, effects and sometimes advice on how to fix them.

## REIMPLEMENTATION OF A LARGE CODEBASE

In order to answer research Questions 2-4, a component of EPICS base, called iocsh (IOC shell), was chosen to be reimplemented in Rust while still being part of the larger C/C++ project, so the difficulties and benefits of Rust could be evaluated.

### *Automatic Translation Attempt*

Before starting to manually translate C/C++ code into Rust, a couple of existing C/C++ to Rust transpilers were tested: CRUST [7] and C2Rust [8].

CRUST was simpler to install and compile, but the program execution "crashed" (panicked, in Rust's parlance) with an "Index out of bounds" error.

C2Rust is a much more complex and interesting project: it leverages LLVM (and therefore, clang) to do analysis and parsing of the original code. It also uses a tool, called BEAR (Build EAR), that has to be used during a normal compilation of the original EPICS code in order to generate some metadata about the build process itself. Then, C2Rust can use the output of the BEAR program to aid in the transpilation. C2Rust itself took 9 minutes to compile. However, it also panicked when used.

Given these results, it seems that automated translation tools from C/C++ to Rust are not yet ready to be used in large codebases. It is not clear that they'll ever be ready for this task, given C, C++ and Rust great complexity and differences.

## REIMPLEMENTATION OF EPICS' IOCSH

iocsh is a simple shell that typically runs inside EPICS IOCs. The responsibilities of iocsh are to parse commands given to it, find the parsed commands in a global registry of available commands, and execute them. This also involves maintaining the global registry of commands and exposing an interface to allow different parts of EPICS to register new commands with iocsh. It has only 1.5 kloc, and sizable chunks of them can be replaced with Rust's standard library functions.

Due to a higher degree of difficulty than anticipated, a non-trivial amount of time was spent attempting different approaches on how to reimplement the shell in a way that preserves functionality and is fully interoperable with C. For example, one obstacle was the fact that the registry that iocsh maintains is implemented, in C, as a program-global hash map and a linked list of structures describing the available commands, which is not protected by a mutual exclusion lock (presumably because the registry is only populated during IOC startup, which is single threaded at that point). Rust ordinarily does not allow write access to non-locked global data, so a different approach was needed.

Another important obstacle was the integration between Rust and C code, in the sense of finding exactly how to link (in the linker sense) both Rust-generated and gcc-generated object files together. Since Rust is still fairly new and rapidly changing there's a lot of conflicting advice, most being obsolete, on how to approach this issue. However, a Rust users forum [9], which seems to be very active, was successfully used to obtain help.

It is worth noting that, in the course of this study, a bug in EPICS itself (that leads to a segmentation fault), was found and reported by the author on EPICS' Issue Tracker. This bug would have been prevented by Rust, and would have been counted in this paper's analysis.

### *Build System Integration*

The EPICS framework has a custom build system, built on top of Makefiles, that is capable of compiling EPICS for several different targets: from the usual Linux/Windows/Mac targets to more niche vxWorks and RTEMS embedded operating systems. Rust has a standalone, Rust-specific build system, named cargo, which manages not only the compilation itself but also project dependencies.

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WECPR02

In order to have EPICS be able to compile the iocsh port to Rust, the Makefile associated with iocsh had to be modified to use cargo. Cargo compiles the Rust code into a C ABI compatible shared object library file, which is then linked to the rest of the unmodified EPICS framework. It is important to note that, as configured for this project, cargo only compiles for the host architecture (Linux on x86_64) and as a *debug* (as opposed to a *release*) target.

With this setup, it is possible to compile both regular EPICS code and the rewritten iocsh module at the same time by simply issuing `make` at the root of the project folder.

### *Global Data Structures*

As mentioned before, iocsh exports functions that allow any part of EPICS to register new commands with the shell. These functions can be called at any time, and the registered commands are kept in a couple of data structures with static storage inside iocsh: a hash map (with a custom C implementation), in order to allow O(1) command lookups, and a linked-list (also with a custom C implementation), in which command description structures are kept in alphabetical order with respect to the command names, presumably to make it easy to display the list of available commands with the help command.

In the Rust version the standard library's `HashMap` data structure was used. In order to display available commands in alphabetical order, the help iocsh command simply sorts the names of the commands in the `HashMap` before printing them to the screen. Since Rust does not like variables that are static, global and mutable, because they are not thread-safe, the global `HashMap` had to be put behind a `RwLock` (Read-Write Lock), so it could be concurrently read by many threads and be written to in a mutually exclusive way. Rust knows that synchronization primitives, such as the `RwLock` , can be used by different threads safely.

### *FFI – Foreign Function Interface*

The EPICS framework provides many programming constructs to its users as a way to fill the gaps in C's sparse standard library and as a way of allowing EPICS programs to be written in a platform-agnostic way. Examples of such constructs are implementations for a general-purpose hash table and for a linked list, as mentioned before. Modern languages like Rust have such constructs available in their standard libraries. However, one interesting facility provided by EPICS is a macro expansion library, called macLib. macLib allows the users of the IOC shell to be able to parameterize commands by making use of macro expansions, as shown in Listing 5.

Listing 5: EPICS macro substitution in an IOC

```
epics> epicsEnvSet("HELLO", "Hello,
world!")
epics> echo $(HELLO)
Hello, world!
```

While it would be perfectly possible to rewrite macLib in Rust, it is much better to be able to just use such readily available functionality. Hence, this was a great opportunity for taking advantage of Rust's binary compatibility with C. To that end, all available functions in macLib were declared in a way that Rust can understand and use them. Thin wrappers around them were created to act as an interface between safe (Rust) and unsafe (C) code, while also performing data type conversions between the languages. One such noteworthy conversion is between C and Rust strings: a C string is essentially a pointer to a region of memory that has a null byte as its terminating character; a Rust string, on the other hand, has length information encoded in them and no terminating null byte. Also, Rust strings are UTF-8 encoded.

Even though the macLib wrapper was very thin, it still amounted to 146 lines of code (as counted by cloc), which speaks to the amount of work needed to craft such wrapper.

### *Command Parser*

The main responsibility of the IOC shell is to receive commands from a user or a script and execute them. However, the syntax for the language that the shell accepts is not formally defined; instead, command line inputs are parsed in an *ad-hoc* way. The IOC shell makes use of the widely-used `libreadline` library to provide command-line editing and history capabilities.

In the Rust reimplementation the parsing of commands is done by making use of regular expressions through the regex crate (a crate, in Rust's parlance, is akin to a library in C or Python). The reimplementation also allows command-line editing and history, via the rustyline crate. The use of regular expressions greatly simplified the code for parsing inputs to the shell, at a loss of more precise error messages.

### *Reverse FFI*

Since the main objective is to rewrite part of a C/C++ project in Rust, the resulting Rust module must be able to communicate with C/C++ modules. Communications in one direction (Rust accessing C functions) were achieved in the use of the macro expansion library, macLib. Communication in the other direction (C accessing Rust functions) is made possible by marking the structures and functions in the Rust module API as C-compatible. This tells the compiler to generate binaries that can be used by C.

This was the bulk of the iocsh reimplementation and its most challenging part. Passing objects back and forth between C and Rust proved to be difficult due to Rust's great strictness about object lifetimes and access rules, contrasted with C's complete lenience. In many instances it was laborious to determine the ownership of certain resources (and, by extension, which language is responsible for freeing them) coming from C. For example, when registering a command with iocsh, EPICS code allocates *static* structures and pass pointers to them to iocsh. iocsh, however, is expected to allocate a new structure on the *heap* that has some more metadata about

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WECPR02



the command being registered, along with pointers to the passed-in static structures. However, nowhere it is specified that the passed in structures have to be static, it is just convention. This kind of implicit lifetime information had to be made explicit to Rust's compiler, which involved a lot of boilerplate and data type conversion code.

## CONCLUSION

Compiled, typed languages present a great opportunity for static analysis tools to be run in order to catch bugs in a program before they occur at runtime. The two major systems languages, C and C++, notably have shortcomings both in their design, preventing the compiler from statically catching certain important classes of bugs such as null pointer de-reference, and in their runtime implementation, allowing for out-of-bounds access of array elements. While it can be argued that the lack of such checks makes C and C++ programs faster, and that modern C++ programmers can mitigate some of these issues by using newer constructs, such as smart pointers, it is a fact that unsafe constructs can still be used, due to retrocompatibility requirements, sometimes without even a warning, as observed in this study.

Big C/C++ projects stand to benefit greatly from Rust's features. In order to evaluate if that would be the case for the EPICS framework, this two-part study was conducted. First, real world EPICS bugs were analyzed, classified and then rewritten in simplified C or C++ and in Rust to compare the GNU and Rust languages and compilers. Then, a single EPICS component, iocsh, was chosen to be reimplemented in Rust while still interfacing with the rest of the EPICS framework. The evaluation of these two studies provided the basis for the answers to the following research questions:

### *Would Rust have Prevented Actual EPICS Bugs?*

Yes. Even though the analysis was done on the simplified versions of the selected bugs, it was clear that the memory safety features of the Rust compiler would have prevented the compilation of such bugs from proceeding, while probably providing good error messages as to why.

From the 185 classified issues, 41 (22.2%) of them were due to memory safety or type casting issues. That corresponds to more than a fifth of issues that could have been prevented if Rust-like safety features were used. Microsoft recently conducted a study [10] which stated that approximately 70% of the security vulnerabilities they address every year are memory safety issues. This underlines the importance of stronger mitigation for such issues.

### *Is it Straightforward to Translate C/C++ Code into Rust?*

Reasonably. Rust has many modern features that can be easily mapped into from modern C++ code. C code is trickier to port since C has no concept of RAII, so object lifetimes have to be figured out first. While Rust lacks some C and C++ low level features, such as bit fields (which is being addressed by the Rust FFI working group), it does have a familiar C-like syntax which makes it easy for more direct reimplementations.

### *Is it Feasible to Rewrite Parts of EPICS into Rust?*

Yes, as was demonstrated by this study, but at a great cost. The most important problem of porting just one component of a C or C++ project into Rust is the interface between the two languages. Since Rust is very strict with memory safety, a *lot* of code has to be written as a glue between the languages to make sure that Rust's assumptions are not being violated. The author of the wlroots-rs Rust crate faced this problem and came to the following conclusion (with their emphasis) [11]:

*"Currently there is 11 THOUSAND lines of Rust in wlroots-rs. All of this code is just wrapper code, it doesn't do anything but memory management. This isn't just repeated code either, I defined a very complicated and ugly macro to try to make it easier. This wrapper code doesn't cover even half of the API surface of wlroots. It's exhausting writing wlroots-rs code, memory management is constantly on my mind because that's the whole purpose of the library. It's a very boring problem and it's always at odds with usability"*

If the lifetime and ownership of the objects being passed between the languages are well defined (which is often not the case), this task becomes somewhat easier, if not repetitive. In any case, careful consideration must be put into deciding if the safety guarantees that Rust provides are worth the cost of rewriting specific components into Rust. Maybe this is true for core, critical components, but no so much for peripheral components.

### *Is it Worth it to Rewrite a Big C/C++ Project into Rust?*

Probably not. Big C and C++ projects usually have a long history of use and have had many bugs squatted. Rewriting code into another language invariably introduces new bugs. New components to a big C/C++ project, however, could potentially be written in Rust, if the interface between the two languages in the project is well defined. Additionally, as stated before, it might be more cost-effective to rewrite only the critical components of a project in Rust, rather than the *entire* project. New systems-level projects, however, would benefit greatly from using Rust from the start.

## REFERENCES

[1] N. D. Matsakis and F. S. Klock II, "The Rust Language," in *Proc. 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, New York, NY, USA, 2014, pp. 103–104, doi:10.1145/2692956.2663188

[2] J. Triplett, "Intel and Rust: the Future of Systems Programming," presented at the Open Source Technology Summit, Stevenson, Washington, USA, 2019.

[3] EPICS, https://www.epics-controls.org

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WECPR02

[4] Bugs: EPICS Base, https://bugs.launchpad.net/epics-base

[5] T. Hoare, "Null References: The Billion Dollar Mistake," presented at the QCon London, London, UK, 2009.

[6] The Rust Programming Language, https://doc.rust-lang.org/book

[7] CRUST, https://github.com/NishanthSpShetty/crust

[8] C2Rust, https://c2rust.com

[9] The Rust Programming Language Forum, https://users.rust-lang.org

[10] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," presented at BlueHat IL 2019, Tel Aviv, Israel, 2019.

[11] Giving up on wlroots-rs, http://wayooler.org/blog/2019/04/29/rewriting-way-cooler-in-c.html